Université de Montréal

Compilation efficace pour FPGA reconfigurable dynamiquement

par

Etienne Bergeron

Département d'informatique et
de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
philosophiæ doctor (Ph.D.)
en informatique

Août 2008

Université de Montréal
Faculté des études supérieures et postdoctorales

Cette thèse intitulée :
Compilation efficace pour FPGA reconfigurable dynamiquement

présentée par :
Etienne Bergeron

a été évaluée par un jury composé des personnes suivantes :

Pierre Poulin
président rapporteur

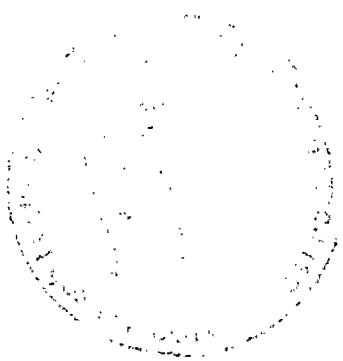
Jean Pierre David
directeur de recherche

Marc Feeley
codirecteur

Stefan Monnier
membre du jury

Claude Thibeault
examinateur externe

Anne Bourlioux
représentant du doyen de la FESP



Thèse acceptée le : 23 Décembre 2008

Sommaire

Mots-clés : Synthèse dynamique, JIT, VirtexII-Pro, bitstream, FPGA

La combinaison de l'évaluation partielle et de la génération de code à la volée forment un mode d'exécution dont les gains ont déjà été démontrés en logiciel par leur utilisation dans les compilateurs JIT (*Just-in-Time*). Le domaine matériel tarde cependant à suivre le pas, malgré les avancées technologiques réalisées ces dernières années en matière de reconfiguration dynamique.

Un tel retard dans l'intégration de ces optimisations classiques par rapport au domaine logiciel s'explique par les défis algorithmiques et techniques importants sous-jacents à leur déploiement sur une architecture matérielle.

Les modes d'exécution que nous proposons tirent profit de la restructuration dynamique du support d'exécution et reposent de ce fait sur une architecture configurable. Le premier défi posé par l'implantation d'une telle optimisation est le manque d'outils permettant l'intégration de ces modes d'exécution au flux de développement matériel. Afin de pallier cette lacune, de nouveaux outils ont été réalisés dans la présente recherche.

Le second défi posé réside dans le manque de support pour la manipulation des configurations matérielles, tant au niveau des outils que de la documentation. Afin de combler ce manque, nous avons créé une bibliothèque compacte permettant d'adapter l'architecture en modifiant sa configuration dynamiquement, ce qui permet l'implantation de nouvelles classes d'applications. La réalisation d'un compilateur JIT prouve l'utilisabilité de l'approche que nous proposons, ouvrant ainsi la voie à de nouveaux axes de recherche.

Abstract

Keywords : Dynamic synthesis, JIT, VirtexII-Pro, bitstream, FPGA

Combination of partial evaluation and on-the-fly code generation form an execution mode of which gains have already been demonstrated in the software field by their use in JIT (Just-in-Time) compilers. The hardware field however defers to follow, despite technological progress have done in the past years in terms of dynamic reconfiguration.

Such a delay in integrating those classical optimisations with respect to the software field can be explained by the important algorithmic and technological challenges underlying their deployment on a hardware architecture.

The execution modes we propose benefit from dynamic restructuring of the execution support and depend on a configurable architecture. The first challenge faced when implementing such an optimisation is the lack of tools allowing integration of these execution modes to the hardware development flow. To overcome this shortcoming, new tools have been built in the research hereby presented.

The second challenge comes from the lack of support for hardware configuration manipulation, both at the tool and documentation levels. To fill this gap, we have created a compact library allowing the adaptation of the architecture by modifying its configuration dynamically, which permits to implement new application classes. The creation of a JIT compiler proves the usability of the approach we propose, therefore opening the path to new research areas.

Table des matières

Table des figures

Liste des tableaux

Abbréviations

ASIC	Application Specific Integrated Circuit
APU	Auxiliary Processing Unit
CLB	Configurable Logic Block
DES	Data Encryption Standard
DMA	Direct Memory Access
DSP	Digital Signal Processing
EP	Évaluation Partielle (*Partial Evaluation*)
ER	Exécution Reconfigurable (*Reconfigurable Computing*)
FPGA	Field Programmable Gate Arrays
FSM	Finite States Machine
ICAP	Internal Configuration Access Port
IOB	Input Output Block
JIT	Just-in-Time
LUT	Look-up Table
MD5	Message Digest 5
OCM	On-Chip Memory
OPB	On-Chip Peripheral Bus
PCI	Peripheral Component Interconnect
PIP	Programmable Interconnect Point
PLB	Processor Local Bus
PRNG	Pseudorandom Number Generator
RAM	Random Access Memory
UCF	User Constraints File
SOC	System On Chip

Publications

Cette section contient les publications contenues dans cette thèse. Toutes les publications sont déjà parues, ou sont acceptées et en cours de parution.

1) *Logarithmic Time FPGA Bitstream Analysis : a Step Towards JIT Hardware Compilation*
Etienne Bergeron, Louis David Perron, Marc Feeley, Jean Pierre David
Journal ACM Transactions on Reconfigurable Technology and Systems

2) *High level synthesis for data-driven applications*
Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley, Jean Pierre David
Rapid System Prototyping, 2005. (RSP 2005)

3) *Toward on-chip JIT synthesis on Xilinx VirtexII-Pro FPGAs*
Etienne Bergeron, Marc Feeley, Jean Pierre David
Circuits and Systems, NEWCAS 2007 (5-8 Aug. 2007)

4) *Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs*
Etienne Bergeron, Marc Feeley, Jean Pierre David
Compiler Construction, (CC 2008). Lecture Notes in Computer Science

5) *Using Dynamic Reconfiguration to Implement High-Resolution Programmable Delays on an FPGA*
Etienne Bergeron, Marc-Andre Daigneault, Marc Feeley, Jean Pierre David
System-level Design, NEWCAS 2008 (22-25 June 2008)

La réalisation d'un système complexe a été possible grâce à collaboration avec d'autres personnes. Nous résumons leurs apports à nos publications.

Louis David Perron est intervenu dans le cadre d'un projet d'été à l'Université de Montréal. Il a complété l'information manquante à l'algorithme de correspondance inversse pour déterminer l'encodage de plusieurs points de configuration manquants. De plus, il a résolu de nombreuses contraintes de dépendance non documentées par Xilinx. Par la suite, il a utilisé l'information obtenue pour réaliser un prototype de simulateur de *bitstream*.

Xavier Saint-Mleux a développé un compilateur de Scheme vers matériel. Il a étudié les techniques de synthèse de haut niveau et complété sa maîtrise avec le compilateur SHard.

Marc-André Daigneault, un étudiant de l'École Polytechnique de Montréal, a contribué au développement de la ligne à délai reconfigurable dynamiquement. Il a contribué au prototypage du système et continue le projet dans le cadre de sa maitrise.

Catherine Gaudron, présente dans le cadre d'un cours d'été, a complèté l'outil dynaroute pour corriger les informations manquantes dans les rapports XDL générés par Xilinx. L'information obtenue a permis l'ajout du support des blocs de RAM à la bibliothèque de reconfiguration dynamique.

Remerciements

Je tiens tout d'abord à remercier le directeur de cette thèse, Jean Pierre David, et ainsi que le co-directeur, Marc Feeley, pour m'avoir fait confiance sur un projet qui était audacieux. Ils ont su me guider, m'encourager, me conseiller, me laisser voyager tout en me laissant une grande liberté.

Je remercie du fond du coeur la femme de ma vie, Annie, qui a su me donner tout le support moral (et physique) requis pour traverser les épreuves rencontrées lors de mes péripéties. Annie a su gérer la vie familiale avec un éternel étudiant et deux enfants remplis d'énergie.

Je remercie également mes parents, sans qui je n'aurais pas vu le jour, et par conséquent, cette thèse non plus. Je ne compte plus le nombre de lectures et corrections de "fôtes" corrigées par eux lors de mes publications.

Je remercie Claude et Johanne qui ont été présents tout au long de mes études.

Je passe ensuite une dédicace spéciale à tous les gens qui ont contribué au projet dans le cadre de leurs études, à savoir Louis David Perron, Xavier Saint-Mleux, Marc-André Daigneault et Catherine Gaudron.

Je remercie également Nicolas Gorse, Michel Metzger, Mathieu Desnoyers et Eric Lesage pour leurs nombreuses relectures de mes articles.

Je remercie Pascal Bergeron, Nicolas Bergeron, Pascal Lamblin, Adrien Pierard et Mathieu Desnoyers pour leur financement symbolique, qui pourra probablement me payer une bière après mes études.

Et finalement, je remercie tous ceux qui lisent les remerciements.

Chapitre 1

Introduction

Une application de traitement de données est généralement exprimée par le développeur à l'aide d'un langage de programmation et cristallisée dans un code source, lequel est transformé par un compilateur en une séquence d'instructions qu'un processeur conventionnel peut exécuter. Ce procédé de développement d'applications est populaire puisqu'il possède un avantage considérable : l'architecture matérielle est capable d'exécuter différents programmes sans être modifiée. Cette polyvalence a fait le succès de l'ordinateur personnel tel que nous le connaissons aujourd'hui.

La polyvalence des processeurs conventionnels engendre toutefois un coût de performance pour certaines classes d'applications comme le traitement continu de signaux ou le calcul de prévisions météorologiques. Ces applications effectuent répétitivement une même tâche, nécessitent une puissance de calcul considérable, et elles peuvent typiquement tirer profit d'une architecture adaptée à leurs besoins. Par exemple, un processeur DSP (*Digital Signal Processing*) qui est optimisé pour effectuer des calculs complexes (e.g. *Multiply and Accumulate*) en un cycle d'horloge et pour accéder à un grand nombre d'entrées-sorties (numériques ou analogiques) accélère le traitement de signal. Ainsi, nous pouvons dire que certaines architectures matérielles doivent être adaptées pour permettre à ces applications d'obtenir les performances requises.

Une façon de parvenir à une efficacité accrue est de réaliser une architecture spécifique. Les circuits intégrés spécifiques (*ASIC*) sont conçus dans l'optique d'atteindre de bonnes performances. Ils ne peuvent cependant pas exécuter une application autre que celle ciblée initialement.

De plus, le développement de telles applications est laborieux et nécessite des concepteurs spécialisés. En conséquence, ceci limite le nombre et le type d'applications pouvant tirer profit de l'accélération matérielle car il faut amortir les coûts de développement sur les grands marchés.

Un compromis intéressant entre les deux modes d'exécution est d'utiliser une architecture reconfigurable. L'exécution reconfigurable (*reconfigurable computing*) consiste à spécialiser une architecture configurable pour l'adapter aux besoins de l'application dans le but d'améliorer ses performances. Elle donne aux logiciels la flexibilité des processeurs conventionnels et les performances des circuits intégrés spécifiques [GNVV04]. Ce processus de développement est néanmoins peu accessible vu le besoin de connaissances approfondies de l'architecture reconfigurable et le manque d'outils adaptés. En pratique, cette approche n'est pas suffisament employée même si plusieurs applications pourraient en tirer profit.

Typiquement, l'exécution d'une application matérielle utilisant la reconfiguration dynamique se limite aux chargements et déchargements dynamiques de certaines parties de son code. Puisque ces parties sont compilées avant toute exécution, ce modèle ne permet pas de les spécialiser selon les propriétés dynamiquement observées. La spécialisation de modules matériels permet d'augmenter la densité fonctionnelle de l'application et augmente ainsi son efficacité.

L'entraînement d'un réseau de neurones est un exemple d'application qui a su tirer profit de la spécialisation dynamique [EH94a]. L'application contient un réseau de multiplications et d'additions dans lequel chaque multiplication possède une entrée considérée constante pour la durée de l'entraînement d'un lot. Le temps de synthèse est absorbé par l'utilisation répétitive du module matériel spécialisé.

Le temps de synthèse élevé ne peut pas être absorbé par certaines classes d'applications et empêche l'application de ce mode d'exécution. Une diminution du temps de synthèse permettrait à d'autres applications d'absorber le temps de synthèse dynamique et donc d'utiliser la spécialisation dynamique de module matériel.

Dans le but d'élargir la classe d'applications pouvant bénéficier de ce mode d'exécution, l'objectif de notre recherche consiste à développer les techniques et les outils nécessaires pour l'exécution reconfigurable qui simplifient la tâche de conception et rendent accessibles les performances du matériel. Nous nous sommes inspirés de la technique de la compilation dynamique (JIT : *Just-in-Time*) pour créer une variante qui cible la synthèse de matériel. La compilation dynamique est transparente, permettant aux développeurs logiciels de réaliser des applications

sans connaître le modèle dynamique qui exécute l'application. Il en est de même pour notre approche.

Les FPGAs (*Field Programmable Gate Arrays*) sont des composantes matérielles reconfigurables qui, par un processus de configuration, peuvent implanter un circuit numérique. Le coût de développement relativement faible et leur grande capacité rendent ces composantes attrayantes pour le développement d'applications matérielles. De plus, les plus récents FPGAs permettent la reconfiguration dynamique, et donc la modification pendant l'exécution du circuit implanté ou d'une partie du circuit. Cette nouvelle fonctionnalité permet maintenant de modifier dynamiquement un circuit pour mieux l'adapter aux besoins de l'application. La technologie actuelle des FPGAs nous porte à croire que, dans les prochaines années, ce modèle d'exécution pourrait s'avérer une solution de remplacement efficace et rentable pour certaines classes d'applications matérielles et faciliterait leur développement. Tout porte à croire que le matériel informatique d'usage général du futur utilisera des parties reconfigurables pour accélérer des calculs.

Pour démontrer la viabilité de la synthèse dynamique sur FPGA, nous avons conçu une architecture reconfigurable reposant sur un FPGA et construit un compilateur capable de produire dynamiquement des configurations pour cette architecture. Nous avons intégré les mécanismes de compilation dans un environnement d'exécution d'un langage de haut niveau pour obtenir un environnement complet de développement d'applications dynamiques. Il est possible pour un utilisateur de spécialiser en cours d'exécution l'architecture selon les besoins de son application sans autre connaissance que le langage de haut niveau.

Les temps de synthèse obtenus démontrent que la compilation efficace pour un FPGA reconfigurable dynamiquement n'est pas le facteur qui limite la spécialisation dynamique. Auparavant, les temps de reconfiguration était négligeables par rapport à ceux élevés de la synthèse. Nous démontrons que le temps de la synthèse et le temps de reconfiguration sont sensiblement les mêmes et que toute amélioration sur les mécanismes de reconfiguration avantagerait la spécialisation dynamique.

1.1 Définitions des termes

Nous appelons **application matérielle** tout programme de la classe d'applications typiquement implantées en matériel. Ces applications se caractérisent par un comportement répétitif

ou par phase nécessitant une vitesse de calcul élevée. Notons que ces applications interagissent peu avec l'usager et orientent ainsi le développement vers l'obtention de performance. Les filtres d'images, le décodage et la décompression de flux de données, le cryptage et le décryptage de données sont des applications représentatives de cette classe.

Nous appelons **composante matérielle** (*hardware device*) une pièce physique ou un circuit électronique contenant de la logique digitale (typiquement une puce). Un **module matériel** est une partie d'une application matérielle. Une composante matérielle est constituée d'un ensemble de modules matériels et notre définition de module est similaire à la notion de *entity* en VHDL et *module* en Verilog. Ainsi, un **module dynamique** est un module matériel qui peut être ajouté ou enlevé à la volée à l'ensemble des modules actifs. Ces modules peuvent être produits statiquement par des outils de développement avant l'exécution, ou dynamiquement pendant l'exécution de l'application.

Nous définissons **configurer** par l'action de fixer des paramètres. Typiquement, nous parlons de la configuration d'une architecture. Ainsi, nous définissons **configuration statique** par une configuration avant toute exécution d'application sur l'architecture, et **configuration dynamique** par une configuration pendant l'exécution d'une application sur l'architecture. Nous définissons **reconfigurabilité** par la propriété de permettre le changement de configuration.

Une **configuration** est une correspondance entre les points de configuration et les valeurs assignées. L'encodage d'une configuration produit un **bitstream** (une séquence de bits). Le chargement de cette séquence de bits dans un FPGA configure le matériel pour implanter le comportement voulu.

L'**exécution reconfigurable** (*reconfigurable computing*) est l'exécution d'une application (matérielle ou logicielle) sur une architecture configurable. Lors d'une exécution reconfigurable, le support de l'exécution est modifié (adapté) pour mieux répondre aux besoins du programme.

Un **compilateur dynamique** est un compilateur par phase (*stage compiler*) dont les dernières phases telles que la génération de code s'effectuent pendant l'exécution de l'application. Par opposition, nous définissons **compilateur statique** par un compilateur dont toutes les phases de compilation s'effectuent avant toute exécution de l'application.

L'**évaluation partielle** est le processus de spécialisation d'un programme (module) à partir des valeurs de certaines variables connues à la compilation.

1.2 Contributions

L'objectif principal de nos travaux de recherche est de démontrer la faisabilité d'un compilateur dynamique qui cible une architecture reconfigurable à grain fin. Pour y parvenir, nous démontrons que les temps de synthèse peuvent être de plusieurs ordres de grandeur inférieurs aux outils conventionnels de développement d'applications statiques.

Nos contributions sont :

1. Détermination du format du bitstream
2. Bibliothèque et outils pour la configuration dynamique
3. Synthèse dynamique rapide

La première limite technologique rencontréeake est le manque de documentation sur le format du bitstream pour les FPGAs disponibles commercialement. Nous avons déterminé une méthodologie pour retrouver l'information manquante et réussi à appliquer cette méthodologie sur les Virtex-II Pro et Spartan-3 de Xilinx. La méthodologie est publiée dans notre article intitulé *Logarithmic Time FPGA Bitstream Analysis : a Step Towards JIT Hardware Compilation* et accepté dans *Journal ACM Transactions on Reconfigurable Technology and Systems* en 2008.

La seconde limite technologique est le manque d'outils pour supporter la reconfiguration dynamique. Nous avons réalisé une bibliothèque supportant la reconfiguration dynamique pour les FPGAs de Xilinx. À l'aide de cette bibliothèque, nous avons réalisé des outils permettant de développer des applications dynamiques. Puis, nous avons démontré l'utilité de cette bibliothèque et des outils en produisant des applications utilisant la reconfiguration dynamique. Notre article intitulé *Using Dynamic Reconfiguration to Implement High-Resolution Programmable Delays on an FPGA*, publié à *NEWCAS* en 2008, démontre une utilisation innovatrice de la reconfiguration dynamique.

Par la suite, nous démontrons que les temps de synthèse peuvent être de l'ordre de grandeur requis par la synthèse dynamique. Notre article intitulé *Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs*, publié à *Compiler Conference* en 2008, démontre qu'il est possible de produire des configurations pour les FPGAs actuels dans des temps requis pour la synthèse dynamique.

1.3 Structure du document

La structure de ce document est une thèse par articles à laquelle nous avons ajouté des détails que nous considérons importants pour comprendre notre recherche.

Le Chapitre 2 contient l'état de l'art. Notre projet joint différentes disciplines de l'informatique : la compilation, la synthèse matérielle et les architectures reconfigurables. Nous avons résumé les projets importants qui nous ont guidés vers notre contribution.

Le Chapitre 3 traite de la méthodologie utilisée pour démontrer que la compilation dynamique d'applications vers des FPGAs reconfigurables dynamiquement est viable. L'évolution du projet et les problèmes rencontrés y sont discutés.

Le Chapitre 4 détaille l'architecture matérielle produite. Nous y énumérons les problèmes rencontrés par ce type de développement et les solutions innovatrices que nous avons utilisées.

Le Chapitre 5 explique les techniques utilisées pour contourner les limites actuelles relatives à la production de configurations. Nous y démontrons l'utilité de la bibliothèque de reconfiguration dynamique que nous avons développée pour réaliser nos applications dynamiques.

Le Chapitre 6 porte sur les détails de Dynabit, le compilateur dynamique que nous avons réalisé.

Le Chapitre 7 contient les performances obtenues par notre système. Les performances de différentes applications y sont étudiées pour évaluer la viabilité de ce type d'exécution. Nous expliquons les points forts et les points faibles de notre système, puis nous tentons de mettre en évidence les améliorations requises pour avoir un système robuste.

Le Chapitre 8 mentionne des perspectives de synthèse de haut niveau qui pourraient être intégrées à Dynabit pour obtenir un système capable de compiler des applications complexes vers le matériel.

Chapitre 2

État de l'art

La compilation est généralement définie comme étant la transformation d'un programme d'un langage informatique ou d'une représentation intermédiaire vers une représentation interprétable ou exécutable par la machine hôte. L'apparition des compilateurs dynamiques (*JIT : Just-in-Time*) a amené un nouveau mode d'exécution hybride qui consiste à effectuer la transformation dynamiquement, c'est-à-dire pendant l'exécution.

Similairement, dans le domaine du matériel, la synthèse consiste en la transformation d'une description matérielle vers son implantation physique (circuit logique). Avec la venue des composantes matérielles et des architectures reconfigurables dynamiquement, il est devenu possible de modifier dynamiquement le matériel pour l'adapter aux besoins de l'application. Les concepts de compilation dynamique sont devenus applicables à la synthèse. Malheureusement, les algorithmes de synthèse sont encore inappropriés à la synthèse dynamique.

Dans ce chapitre, nous décrivons l'évolution de la compilation dynamique et de la synthèse dynamique puisque leur évolution est similaire et que souvent les techniques sont communes. Par la suite, nous décrivons l'exécution reconfigurable (ER, *reconfigurable computing*) et le matériel virtuel qui sont des concepts requis pour comprendre le mode d'exécution d'une application matérielle utilisant la synthèse dynamique. Puis, nous présentons les algorithmes classiques et les algorithmes rapides permettant d'effectuer les différentes phases de la synthèse.

2.1 Compilation dynamique

Pour être exécutée, une application doit être compilée ou interprétée. La compilation d'une application amortit le coût relativement élevé des analyses et des optimisations puisqu'elle n'est effectuée qu'une seule fois pour plusieurs exécutions et qu'elle diminue le coût de la boucle d'interprétation (lecture-décodage-exécution). En revanche, à cause de son caractère statique, elle possède un niveau de flexibilité limité. L'interprétation entraîne un coût d'exécution supplémentaire mais permet des comportements dynamiques intéressants (e.g. chargement dynamique de code). De nos jours, des modèles hybrides utilisent les avantages des deux techniques pour obtenir flexibilité et efficacité.

L'évolution de la compilation dynamique s'est passée en plusieurs phases. La première phase fut le retour de l'utilisation de code modifiable dynamiquement. Par la suite, lors de la seconde phase, un grand nombre de techniques et d'outils ont permis la spécialisation dynamique de code. Certains de ces outils ont continué d'évoluer au point d'être aujourd'hui automatiques et transparents : c'était le début des systèmes avec compilateurs dynamiques. On peut séparer les compilateurs dynamiques de cette troisième phase en deux classes en fonction du nombre de fois et du moment où les recompilations se produisent. Nous décrivons ci-dessous les phases de cette évolution.

2.1.1 Génération dynamique de code

Le concept de production de code à la volée n'est pas récent et a été utilisé à l'origine pour accélérer la recherche de *patterns* à l'aide d'automates spécialisés [Tho68].

Dans les années 90, Keppel [KEH91] et al. ont proposé de revenir vers un modèle où le code exécutable est considéré comme des données et peut, par conséquent, être modifié pendant l'exécution. Ils définissent la génération de code dynamique comme l'ajout de code exécutable au programme pendant son exécution. Keppel et al. donnent deux arguments pour revenir vers un modèle d'exécution dynamique : la portabilité et la spécialisation à la volée. La prolifération de différentes architectures impose, de fait, la portabilité comme un critère majeur dans le développement et une application doit facilement s'adapter dynamiquement à plusieurs architectures d'une même famille. De plus, une application peut produire, à l'aide de générateurs de code, des spécialisations de son code afin d'améliorer ses performances vis-à-vis des coûts associés aux

accès mémoire et les pénalités associées aux antémémoires (*caches*) variables selon les différentes architectures d'une même famille. Une application peut ainsi adapter son code aux propriétés de l'architecture hôte.

Des applications performantes utilisant le concept de génération dynamique de code ont vu le jour. Par exemple, la génération dynamique de code permet d'obtenir des résultats plus efficaces pour une requête dans une base de données [CAK+81] si le code est compilé dynamiquement avant le début de la recherche que s'il est interprété pour chaque entrée de la base de données. La machine virtuelle de Smalltalk80 [Mir87] possède du code statique et du code généré dynamiquement et elle améliore les performances de l'ordre de 40% par rapport au code complètement statique. Chambers et al. ont produit une implantation du langage SELF [CUL89], un langage inspiré de Smalltalk80, qui produit dynamiquement de multiples versions des méthodes et a réussi à en doubler la vitesse d'exécution.

2.1.2 Spécialisation dynamique de code

La spécialisation d'un code est une transformation de programme qui produit une version du code spécialisée selon certains paramètres fréquents. La spécialisation *dynamique* consiste à spécialiser certaines régions du programme selon l'état dynamique. Toutefois, il n'est pas possible de déterminer toutes les spécialisations nécessaires pour l'exécution d'une application, car elles sont généralement trop nombreuses. Par conséquent, la spécialisation dynamique de code est souvent produite à partir de patrons (*templates*) [PLR85, KEH91] de code contenant des "trous" à remplir dynamiquement ce qui diminue le coût de spécialisation. Le système instancie ces fragments de code préalablement préparés et générés à la main ou à l'aide d'outils de compilation et les ajoute au code exécutable actif de l'application.

Leone et Lee ont proposé le concept de compilation reportée [LL93] et implanté ce concept dans le générateur de code Fabius [LL94, LL96]. L'idée est de produire des générateurs de code spécialisés pour chaque région dynamique.

Typiquement, les outils effectuant de la spécialisation dynamique de code n'effectuent ni analyses ni optimisations. Ils fournissent les mécanismes permettant de spécialiser du code et l'utilisateur est responsable de la gestion des spécialisations via une interface.

L'évaluation partielle (EP) [JGS93] est une technique de spécialisation qui consiste à évaluer

les parties de code dont les valeurs (entrées) sont connues au moment de l'évaluation partielle. L'évaluateur partiel, appelé *mix* pour des raisons historiques, est défini comme suit : $mix([\![f]\!], x)(y) = f_x(y)$ où $[\![f]\!]$ représente un code et x un argument constant. L'évaluateur partiel *mix* produit donc à partir de $[\![f]\!]$ et x la fonction F_x, une version de f. L'EP est souvent employée comme mécanisme d'optimisation dynamique de code. Keppel et al. [KEH93] ont proposé des techniques de spécialisation basées sur des expressions constantes lors de l'exécution. Par exemple, un fragment de code effectuant une multiplication avec une constante de façon répétitive peut être spécialisé [BH94].

Keppel et al. [KEH93] ont aussi exploré le domaine de la génération dynamique de code et ils ont défini les coûts permettant de déterminer quand utiliser ou non la spécialisation dynamique. Leur travail a été poursuivi par Auslander et son équipe [GMP+97, APC+96] dans le développement d'outils. Ils ont développé un compilateur C qui génère des *templates* pré-optimisés contenant des "trous" pour recevoir les constantes.

2.1.3 Compilateur dynamique

Les compilateurs dynamiques produisent à la volée des fragments de code spécialisés à exécuter sur l'architecture hôte. Les fragments spécialisés s'adaptent à l'architecture hôte et au comportement réel (dynamique) de l'application. Contrairement aux outils de génération dynamique de code, les compilateurs dynamiques sont transparents et automatiques. L'avantage de cette compilation est de permettre au système d'effectuer des optimisations plus spécifiques à l'architecture ou au comportement évolutif de l'application. Le compilateur repousse certaines étapes de la compilation pour permettre la visibilité de certaines propriétés qui ne sont pas accessibles statiquement. Comme la plupart des propriétés intéressantes sont coûteuses à déterminer ou non décidables, le principal inconvénient à la compilation dynamique est son coût additionnel associé aux mécanismes de compilation et de détection.

Le choix des optimisations est crucial afin de ne pas détériorer les performances du système. Le système doit aussi détecter quelles portions du programme doivent être compilées et à quels moments, pour recueillir assez d'information dynamique. Il doit effectuer un niveau acceptable d'analyses et d'optimisations qui peuvent être conservatrices ou agressives : une optimisation conservatrice consomme peu de temps d'analyse mais génère peu de gains alors qu'une optimisation agressive engendre un coût plus élevé de traitement mais, idéalement, augmente sub-

stantiellement le gain. De plus, le système doit posséder des mécanismes pour échantillonner de l'information dynamique afin d'enrichir les analyses, et cibler les bonnes parties du programme afin de minimiser leur impact sur les performances et augmenter la qualité de l'information recueillie. Le succès des performances d'un tel système dépend de la précision du choix de ces différentes propriétés.

Dynamo [LD97, BDB00] est un système d'optimisation dynamique qui interprète et compile dynamiquement du code pour HP PA-8000. Puisque son langage source est le langage binaire exécutable directement sur un PA-8000, Dynamo est un système complètement transparent. Un fait notablement surprenant lié à ce projet est que certaines applications étaient plus efficaces lorsqu'interprétées par Dynamo que lors de leur exécution native.

Le système Replay [Mat00], tout comme Dynamo, effectue de la compilation dynamique de code binaire. En revanche, au lieu d'interpréter le code et de détecter les parties fréquemment exécutées, il exécute le code sur du matériel spécialisé détectant ces parties.

Compilation sur demande

Les compilateurs de la première génération effectuent une seule compilation du code lors du chargement de l'application, au moment de l'édition des liens ou à l'utilisation spécifique de la partie de code. Chaque partie de code n'est compilée qu'une seule fois. Cette génération de compilateurs, appelés JIT (*Just-in-Time*), tire profit des informations de l'architecture pour produire du code spécialisé. Le compilateur Java Kaffe [Pic] est un compilateur de la première génération. Il effectue la compilation d'un code virtuel (appelé *icode*) vers l'architecture hôte. Ainsi, il permet le caractère dynamique de Java et les performances d'une application compilée. Jalapeño [BCF+99] est une machine virtuelle Java développée chez IBM Research. Depuis, le système est disponible sous le nom de Jikes et en code ouvert (*open source*). L'approche de Jalapeño est de n'exécuter que du code compilé. Il n'y a pas d'interprétation de code.

Optimisations dynamiques

Les compilateurs dynamiques de la deuxième génération effectuent de la recompilation dynamique de fragments de code. Un même fragment de code peut être recompilé plusieurs fois pour se spécialiser selon les nouvelles propriétés dynamiques détectées par le système. Il peut

même exister plusieurs versions du même fragment selon les contextes d'utilisation. La machine virtuelle Hotspot [Sun99, Sun02] contient un tel compilateur dynamique de deuxième génération où les fragments de code peuvent être spécialisés selon les besoins et les ressources actuelles.

2.2 Exécution reconfigurable

L'**ER** [VMS97] (exécution reconfigurable, *Reconfigurable Computing*) consiste en l'exécution d'une application dont les ressources matérielles nécessaires à l'exécution ne sont pas statiques et sont configurables. Le concept de l'ER n'est pas récent [EBTB63, Est02], et son objectif est d'obtenir l'efficacité de l'exécution matérielle tout en conservant la flexibilité du logiciel.

La performance d'un système est souvent le critère principal de développement d'une application. Les circuits intégrés dédiés permettent d'exécuter une application efficacement puisqu'il est possible de spécialiser les composantes matérielles à la tâche de l'application. Néanmoins, un circuit intégré ne peut pas être modifié après sa création. Toutefois, l'utilisation d'un code ou d'un micro-code permet alors de modifier le comportement d'un circuit après sa création bien que les composantes matérielles soient figées. Cette solution est plus flexible mais engendre une dégradation des performances. Le choix d'une bonne architecture pour une application donnée découle d'un compromis entre les différentes propriétés de cette architecture.

L'ER nécessite la présence de matériel pouvant être configuré pour implanter une fonctionnalité spécifique à l'application en cours d'exécution. Elle permet de joindre le modèle des microprocesseurs et celui des composantes matérielles programmables pour mettre à profit les forces de chacun. Les concepts fondamentaux et les techniques développées dans le domaine du co-design matériel et logiciel sont fortement liés à ceux de l'ER. Généralement, l'ER est basée sur les SRAM-FPGA reliés à un processeur classique.

2.2.1 Architecture configurable

Pour supporter l'ER, la présence de matériel configurable est nécessaire. Dans cette section, nous décrivons les propriétés des architectures configurables, leur évolution et décrivons certaines de ces architectures.

L'évolution des architectures configurables s'est étalée sur trois générations que nous présen-

tons brièvement, en expliquant les problèmes résolus par chacune.

Première génération

Les architectures de la première génération sont typiquement composées d'un processeur qui exécute du code séquentiel et d'un ou plusieurs FPGAs tirant profit du parallélisme. Dans cette génération, on retrouve les systèmes DECPERLE [VBR+96], PRISM [AS93] ainsi que SPLASH [GHK+90]. On y retrouve aussi des systèmes plus récents comme RPM-2 [DJSM98] et Transmogrifier-2 [LGvI+97].

L'utilisation d'un processeur comme contrôleur engendre un goulot d'étranglement sur la communication entre le FPGA et le processeur puisqu'ils communiquent fréquemment par un bus. Ces architectures ne peuvent pas être configurées dynamiquement ou partiellement et le temps de configuration complète de l'architecture est généralement élevé.

Un autre problème fréquemment rencontré est le réseau trop statique d'interconnexions entre les composantes configurables. La topologie de l'architecture restreint alors la classe d'applications qu'elle cible.

Deuxième génération

L'augmentation du niveau d'intégration a rendu possible la fabrication de systèmes sur une puce (SOC : *System on Chip*). L'avantage de cette évolution est de diminuer les coûts de communication entre le processeur et le FPGA, et de nouvelles architectures naissent selon la granularité requise pour le traitement. Par exemple, l'architecture FIPSOC [FAM+97] et TRUMPET [PJA+99] sont configurables à grain fin tandis que les architectures Garp [HW97, CHW00] et RAW [WTS+97] sont configurables à grain large.

Afin de diminuer le temps de configuration dynamique, DeHon et ses collègues proposent des changements rapides de contextes avec le DPGA [TCE+95, DeH94, Deh96]. Xilinx propose une autre approche qui consiste à ajouter aux composantes la configuration partielle. Les FPGAs de la famille des Virtex supportent maintenant la configuration partielle dans le but de diminuer les coûts de configuration.

L'architecture SPLASH-2 [ABD92] permet de configurer le réseau d'interconnexions : sur

chaque carte, un FPGA est responsable du routage des données, ce qui permet de spécialiser la logique de routage.

Le problème des architectures de cette génération est l'incompatibilité des applications entre les générations d'architectures. Les applications développées sont fortement dépendantes de l'architecture et de ses contraintes physiques. Souvent, même pour des composantes de la même famille mais de capacité ou de vitesse différente, l'application doit être synthétisée de nouveau et parfois adaptée, ce qui engendre toujours un coût considérable.

Troisième génération

La troisième génération est constituée d'architectures qui contiennent une couche d'abstraction sur la quantité de matériel configurable. Le but est de se rapprocher de la virtualisation du matériel. Cette idée s'apparente au concept des pages virtuelles dans un système de mémoire virtuelle. La virtualisation du matériel permet ainsi de faire abstraction de la quantité de matériel disponible.

Les applications par excellence visées par cette génération sont des applications multimédia. L'architecture PIPERENCH [CWG+98, GSB+00, SWM+02] permet de synthétiser des *pipelines* et donne de bonnes performances pour les applications à flux de données telles que le traitement vidéo. L'application s'exécute sur un *pipeline* virtuel dont chaque étage est configuré en un cycle d'horloge.

Pour diminuer la dépendance entre les membres d'une famille et pour permettre l'utilisation d'une quantité plus grande de matériel que celle disponible, l'architecture SCORE [CCH+00] propose la notion de virtualisation. Cette architecture est développée dans le but spécifique de manipuler des flux de données.

2.2.2 Générateurs dynamiques et transparents de modules matériels

L'exécution reconfigurable a gagné en popularité avec la reconfigurabilité dynamique des FPGAs. Nous classons en deux catégories les techniques utilisées. Dans les deux cas, les modules sont échangés lors de l'exécution, mais ce qui les différencie est le moment où le module dynamique est produit. Dans la première catégorie, les modules sont produits statiquement. Le

projet RTR-JVM (voir Section 2.2.3) est un des membres de cette catégorie. Dans la seconde, les modules sont produits dynamiquement. Les processeurs Warp (voir Section 2.2.4) en sont un exemple.

2.2.3 RTR-JVM

Les concepteurs du projet RTR-JVM [GS05] (*Run-time Reconfigurable Java Virtual Machine*) ont développé une architecture pour des calculs de haute performance écrits en Java. L'idée est d'utiliser des FPGAs pour implanter certaines portions de l'application. Ces portions sont extraites et synthétisées automatiquement à la compilation. Lors de l'exécution, un contrôleur est responsable d'échanger dynamiquement les portions présentes sur le FPGA selon le comportement de l'application.

Le système contient un compilateur qui génère une version logicielle (*bytecode*) et une version matérielle (*module*) ainsi qu'une machine virtuelle Java modifiée capable de communiquer avec le matériel. Le compilateur de Java vers matériel utilise l'outil Forge de Xilinx (un compilateur Verilog).

Ce système permet l'exécution reconfigurable à partir du langage Java. Ce type de développement d'applications dynamiques est automatique et requiert peu de nouvelles connaissances pour un développeur d'applications. Le choix des modules dynamiques, le chargement et le déchargement sont gérés automatiquement par le système.

Une des limitations de ce projet est que les modules sont compilés statiquement et ne peuvent pas tirer profit de l'évaluation partielle dynamique.

2.2.4 Warp

Les processeurs Warp [LSV06] rendent la production de modules dynamiques transparente. Le processeur traduit dynamiquement un code binaire vers une configuration matérielle. Cette technique peut être intégrée à un processeur conventionnel possédant de la logique reconfigurable.

Pour réaliser son système, le groupe de recherche a développé son propre FPGA [LV04] ainsi que des compilateurs et des algorithmes de synthèse rapide [LVT04, LVT05]. Ainsi, les algorithmes de placement et de routage peuvent tirer profit de la régularité de leur FPGA. En

revanche, le niveau d'intégration et la qualité du FPGA conçu n'est pas du niveau de ceux disponibles sur le marché.

Le succès du projet Warp a été de démontrer que la compilation dynamique vers le matériel est viable sur les FPGAs modernes.

Toutefois, l'inconvénient du projet est qu'il a demandé la conception d'un FPGA spécifique pour contourner les problèmes actuels reliés à la reconfiguration dynamique de FPGAs. Le matériel développé possède une fréquence d'horloge plus basse, une densité moins élevée, et un coût plus élevé que les FPGAs actuellement disponibles.

2.3 Synthèse rapide

La synthèse matérielle pour FPGA repose sur une suite d'algorithmes complexes pour produire une configuration du FPGA. Le temps d'exécution du processus de synthèse est typiquement élevé : il varie de quelques minutes à plusieurs heures. Différentes raisons motivent la compilation rapide : le prototypage matériel, la compilation dynamique de *netlist* et le coût monétaire de la conception et de la simulation du matériel.

Les longues compilations d'applications matérielles ralentissent le cycle de développement. Au vu de la croissance de la complexité des applications matérielles et de la densité croissante des composantes, le besoin de techniques de compilation rapide se fait sentir par l'industrie et le domaine académique.

Ainsi, des algorithmes plus rapides pour certaines phases ont été développés. On note parmi ceux-ci des modifications aux algorithmes classiques, par exemple la phase de placement qui se fait souvent par un recuit simulé (*simulated annealing*). L'outil VPR [SBR98] propose de modifier la gestion de la température modifiée selon une relation dépendant du nombre de déplacements réussis. Ainsi, l'algorithme passe peu de temps dans les températures qui ne convergent pas vers une solution. Quoique ces techniques diminuent le temps de compilation, elles ne rendent pas possible la compilation dynamique de modules matériels puisque leurs temps d'exécution varie de l'ordre de la minute à parfois des heures.

Pour obtenir des performances acceptables pour la compilation dynamique, de nouvelles techniques doivent être développées. Par exemple, Dehon et ses collègues proposent, avec GAMA [CCDW98],

d'effectuer la correspondance technologique et le placement en temps linéaire pour des applications à flux de données.

Nous n'avons pas encore trouvé d'algorithme efficace pour effectuer les interconnexions. A notre avis, le plus approprié est celui implanté par le projet Warp. L'algorithme de routage de ROCR [LVT04] (*Riverside On-Chip Router*) utilise 13 fois moins de mémoire que VPR et est 10 fois plus rapide. L'ordre de grandeur du temps d'exécution de l'algorithme est celui requis par la synthèse dynamique : environ une seconde pour effectuer les interconnexions de l'algorithme de cryptographie DES.

2.4 Conclusion

La production de code dynamique a évolué au cours des dernières décennies. Des techniques de plus en plus évoluées font de la production dynamique de code un mode d'exécution de plus en plus commun pour les langages modernes. Le domaine du matériel et de la reconfiguration dynamique emboîtent le pas sur ce type d'exécution.

La complexité des problèmes rencontrés par la synthèse est d'un autre ordre de grandeur et nécessite de nouvelles techniques innovatrices. Nous croyons que le gain potentiel d'une exécution matérielle pourrait rendre ce mode d'exécution commun pour plusieurs classes d'applications.

Nous estimons que la synthèse à la volée doit générer des modules dynamiques en moins d'une seconde. La diminution du temps de synthèse ouvre la porte à de nouvelles applications matérielles. Ainsi, plus les améliorations de performances sont importantes, plus grande est la quantité d'applications supportées par notre modèle d'exécution.

La recherche dans ce domaine est sans aucun doute jeune et prometteuse.

Chapitre 3

Méthodologie

Pour démontrer la viabilité de la compilation dynamique sur les FPGAs modernes, nous avons construit le système *Dynabit* qui exécute une application et lui permet d'adapter son architecture pendant l'exécution. Le système produit et configure dynamiquement des instructions spécialisées implantées en matériel dans le but d'accélérer l'application en cours d'exécution.

Le projet se divise en trois parties : l'architecture matérielle, l'environnement d'exécution et le compilateur dynamique. La première partie consiste à construire une architecture matérielle permettant la reconfiguration dynamique depuis une application s'exécutant sur un processeur conventionnel. La seconde partie consiste en un environnement d'exécution capable d'exécuter une application écrite dans un langage de haut niveau. La dernière partie, qui effectue le pont entre les deux autres, consiste en un compilateur dynamique capable de produire à la volée une configuration pour l'architecture hôte à partir du langage de haut niveau.

La construction d'un tel système vise à démontrer qu'il peut être plus efficace de modifier pendant l'exécution l'architecture pour l'adapter aux besoins d'une application selon son état dynamique. De plus, nous voulons démontrer que ce mode d'exécution est réalisable sur les FPGAs présentement disponibles sur le marché.

Dans ce chapitre, nous donnons une vue d'ensemble des parties du projet ainsi que son évolution. Nous détaillons et justifions les choix de développement que nous avons faits pour parvenir au système complet. Les détails des différentes parties du projet sont développés dans les chapitres suivants.

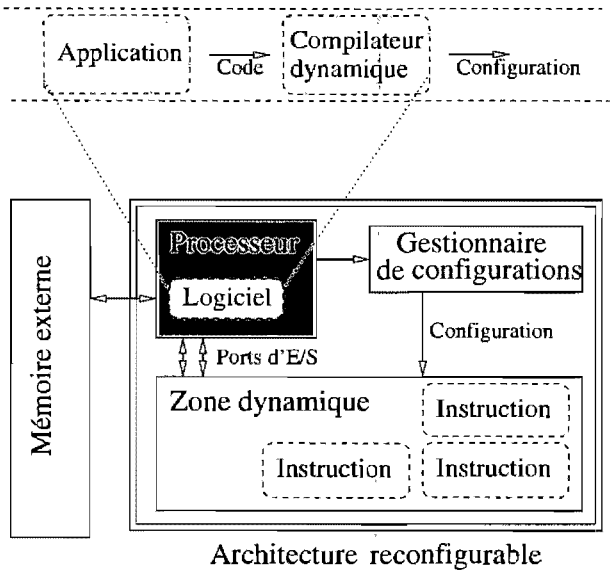


FIG. 3.1 – Architecture de Dynabit

La Figure 3.1 contient le diagramme conceptuel de l'architecture de Dynabit. L'application produite par l'usager s'exécute sur le processeur et utilise le compilateur pour effectuer une compilation à la volée d'une partie de code et obtenir une configuration matérielle. En envoyant cette configuration au gestionnaire de configurations, l'architecture enrichit son ensemble d'instructions avec un nouveau module matériel connecté au bus du processeur. Par la suite, l'application peut communiquer avec le module matériel implantant l'instruction spécialisée au travers de ports d'entrée/sortie. L'objectif est d'exécuter plus rapidement une portion du code (e.g. l'instruction spécialisée) en tirant profit du parallélisme disponible au niveau matériel. Le coût de production à la volée est amorti par l'utilisation répétitive d'instructions plus performantes.

3.1 Architecture matérielle

Pour répondre aux besoins que l'on a fixés à notre projet, nous avons dû utiliser des FPGAs disponibles sur le marché. Les performances de notre système doivent être du même ordre de grandeur que celles obtenues par les processeurs actuels et doivent permettre d'exécuter des applications réelles.

Les besoins sont :

Processeur : un processeur conventionnel et un compilateur C,

Couplage : un couplage étroit avec le processeur,

Fréquence : une fréquence d'opération de la logique comparable à celle des processeurs conventionnels,

Densité et capacité : une capacité élevée pour supporter des applications réelles,

Reconfiguration : support matériel pour la reconfiguration dynamique,

Coût : un coût monétaire raisonnable pour le matériel.

Notre choix de composante matérielle s'est porté vers les FPGAs de la famille VirtexII-Pro de Xilinx qui répondent aux critères recherchés. Ces FPGAs contiennent un ou deux processeurs PowerPC (PPC405) à une fréquence maximale de 400 MHz intégrés à la logique reconfigurable qui, elle, a une fréquence maximale de 100 MHz [1]. Il est possible de connecter le processeur à la logique reconfigurable via le bus du processeur, ce qui permet un bon couplage des modules matériels. Le prix raisonnable et la densité de ces composantes ont aussi guidé notre choix.

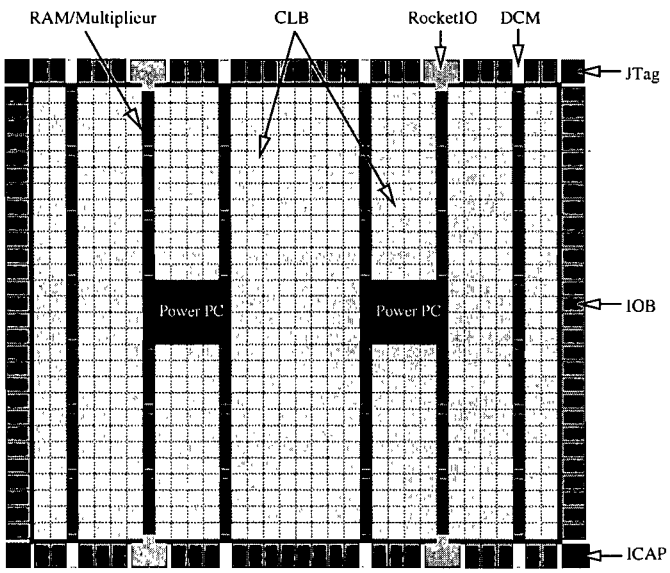


FIG. 3.2 – Organisation physique de la puce du VirtexII-Pro

La Figure 3.2 est une vue simplifiée des parties internes d'un FPGA de la famille des Virtex-II Pro. Les IOBs (*Input/Output Block*) en bordure de la puce sont connectés physiquement à

[1] fréquence maximale permise par l'environnement de développement XPS de Xilinx pour la carte de prototypage ML310 de Xilinx

des broches externes et permettent la communication avec l'extérieur. Les CLBs (*Configurable Logic Block*) sont les blocs de base configurables pouvant implanter de la logique combinatoire et séquentielle. La configuration de ces blocs s'effectue par une communication avec l'ICAP (*Internal Configuration Access Port*) pour modifier les bits de configuration conservés dans une mémoire interne (*SRAM-based FPGA*). Le FPGA contient des blocs de RAM intégrés et peut communiquer avec de la mémoire externe via les IOBs.

Nous avons utilisé la plateforme de développement ML310 commercialisée par Xilinx, qui contient un VirtexII-Pro (VP30) avec une grille de 46 par 80 CLBs. Elle est constituée de plusieurs composantes externes au FPGA (e.g. PCI, port sériel, port parallèle, clavier, ...). Dans notre recherche, nous avons construit un prototype qui n'utilise que le FPGA, la mémoire externe (256 Mo) et le port sériel pour les entrées/sorties de l'application.

Depuis le choix initial du matériel pour cette recherche, de nouvelles familles de FPGAs qui supportent encore mieux la reconfiguration dynamique ont vu le jour : les Virtex-4 et les Virtex-5 de Xilinx. L'implantation de notre projet sur un Virtex-4 ou un Virtex-5 nécessite de construire une nouvelle architecture matérielle, d'ajouter le soutien à notre bibliothèque et d'adapter nos algorithmes de synthèse. Le temps requis pour adapter notre système aux nouveaux FPGAs est trop élevé pour considérer faire ce passage dans le cadre de cette thèse. Cependant les démonstrations réalisées ici nous permettent de croire que le tout se ferait sans difficultés majeures.

Plus de détails sur l'architecture, les problèmes rencontrés et les solutions apportées sont décrits dans le Chapitre 4.

3.2 Environnement d'exécution

L'environnement d'exécution fournit un contexte d'exécution à l'application : ramasse-miettes (*garbage collector*), bibliothèques, chargement dynamique de code, interprétation et exécution de code.

Pour simplifier le développement de notre prototype, nous avons utilisé un environnement d'exécution existant, Gambit-C [Fee], que nous avons adapté à nos besoins. Gambit-C, développé par Feeley, est un compilateur Scheme vers C qui fournit l'environnement nécessaire pour

exécuter des programmes écrits dans le langage Scheme.

Le programme utilisateur et le compilateur Dynabit sont écrits en Scheme et sont compilés avec Gambit-C et GCC que fournit Xilinx afin d'obtenir un binaire qui s'exécute nativement sur le processeur pour PowerPC 405.

Nous avons voulu rendre transparents les concepts introduits par la synthèse dynamique dans la représentation des données de l'environnement d'exécution. Le langage Scheme a été enrichi par une primitive `synthesize` responsable de la compilation dynamique qui prend en paramètre une seule fonction et retourne comme résultat une fonction. L'entrée correspond au code que l'utilisateur souhaite optimiser, et la sortie, idéalement, à sa version matérielle. Dans le cas où la compilation dynamique réussit, la fonction retournée effectue la communication avec le module via les ports de communication. Autrement, la primitive retourne la fonction initialement reçue en paramètre qui exécute le code en logiciel. Pour résumer, `synthesize` est une fonction identité observationnelle sur les fonctions qui spécialise le code en matériel et à la volée lorsque c'est possible.

```
1 (define mult
2   (lambda (x)
3     (lambda (y)
4       (* x y))))
5
6 (pretty-print
7   (map (synthesize (mult 42))
8        '(1 2 3 4 5 6 7 8 9 10)))
```

FIG. 3.3 – Utilisation de la primitive `synthesize`

La Figure 3.3 contient un exemple de l'utilisation de la primitive `synthesize` où la fonction `mult` retourne une fonction (aussi appelée une fermeture ou *closure*) qui multiplie `y`, son paramètre formel, par une constante. L'appel de `synthesize` sur cette fonction configure un multiplicateur spécialisé (multiplicateur par 42) implanté en matériel. L'appel à la fonction `map` envoie une série de données au *pipeline* ainsi généré.

À l'aide de `synthesize`, l'utilisateur est responsable de déterminer quelle portion de code sera compilée ainsi que le moment de sa compilation alors que le système gère automatiquement le chargement et le déchargement des instructions spécialisées.

De cette façon, nul besoin de notions de développement matériel sur un FPGA ou d'archi-

tecture reconfigurable. Comme les primitives du langage Scheme sont reconnues et implantées en matériel, le programmeur peut concentrer ses efforts sur le développement de son application et non sur le développement ou la compréhension de l'architecture matérielle.

3.3 Compilateur

Dynabit transforme une fonction en une configuration matérielle par une suite de phases qui combinent des phases de compilation et des phases de synthèse. La partie frontale du compilateur effectue des analyses et des optimisations classiques (e.g. *lambda lifting*, évaluation partielle, analyse de code mort, ...), et la partie caudale effectue par la suite la synthèse (e.g. correspondance technologique, placement et routage).

Les algorithmes utilisés lors des phases sont les techniques usuelles du domaine. Toutefois, étant donné que le placement et le routage dominent largement le temps requis pour effectuer les autres phases, la plus grande partie de notre recherche s'est portée sur leur accélération.

3.3.1 Évaluation partielle

L'évaluation partielle est une technique de transformation de programme qui consiste à spécialiser un code selon des entrées statiques connues. Essentiellement, les entrées connues sont substituées et les expressions pouvant être évaluées statiquement sont évaluées. Cette phase permet à certaines applications de simplifier de manière significative la taille de certains modules et d'augmenter ainsi la densité fonctionnelle [WH97].

Dans le cadre de notre recherche, cette phase est importante puisqu'elle nous démarque des autres approches et justifie notre besoin d'un générateur à la volée de modules puisqu'il n'est pas possible de générer statiquement l'ensemble des configurations possibles selon les paramètres dynamiques.

3.3.2 Synthèse de haut niveau

La synthèse de haut niveau consiste à déterminer comment implanter en matériel les concepts présents dans le langage de haut niveau, tels la représentation de fermeture en matériel. C'est

à ce niveau que le compilateur décide si une partie du code sera implantée par une machine à états (*FSM : Finite States Machine*) ou par un *pipeline*.

Présentement, Dynabit ne supporte que les expressions simples et produit un *pipeline*. Il ne supporte pas les structures de contrôle itératives ni les fonctions. Les programmes plus évolués sont partiellement évalués jusqu'à ce qu'ils soient représentables par des expressions. Ceci a pour effet de dérouler toutes les boucles et de faire l'expansion des appels de fonction (*inlining*). Aucune notion d'état n'est supportée par le matériel produit par Dynabit.

Le Chapitre 8 introduit certaines améliorations possibles à cette phase qui enrichiraient l'ensemble des concepts du langage de haut niveau supportés par Dynabit. Le sous-ensemble du langage supporté par Dynabit est assez complet pour effectuer de la synthèse dynamique. L'ajout de concepts n'accroît pas substantiellement les phases coûteuses (placement et routage) qui peuvent s'effectuer rapidement.

Le compilateur SHard [SMFD06] a étudié les techniques de production de matériel à partir de code Scheme. Les idées qui ont été développées dans le projet SHard, bien qu'applicables à Dynabit, n'ont pas été intégrées.

3.3.3 Placement et routage

Le placement et le routage sont les deux phases les plus coûteuses de la synthèse. La complexité de ces phases porte à croire que la synthèse rapide n'est pas possible. En effet, les algorithmes classiques ne peuvent être utilisés car leur temps d'exécution élevé est inacceptable dans le cadre de la compilation dynamique.

L'approche utilisée par notre compilateur est d'effectuer le placement rapidement avec un algorithme naïf linéaire, puis de corriger certains placements pour augmenter la probabilité de réussite du routage en enlevant les cas indésirables.

Pour le routage, contrairement aux autres approches, nous acceptons d'avoir un algorithme qui puisse échouer en autant qu'il soit rapide dans les cas où il réussit. L'algorithme est vorace et n'est pas itératif. Ainsi les routes sont construites rapidement par des heuristiques implantées selon nos observations de l'interconnectivité du FPGA. Un mauvais routage n'est pas corrigé et peut empêcher le routage global, ce qui a pour effet de retourner la fermeture originale et

d'exécuter le code en logiciel. D'un certain point de vue, notre système est donc un optimiseur de code.

Vue la quantité élevée d'interconnexions, nous avons émis l'hypothèse que l'algorithme de routage aura peu d'échecs. Certains cas qui ont échoué ont été étudiés manuellement, puis corrigés par des règles de placement favorisant un meilleur routage.

3.3.4 Production de la configuration

La production de la configuration *bitstream* pour les FPGAs de Xilinx n'est possible qu'à partir de leur générateur de configuration (*Bitgen*). Des outils tels que PARBIT [HLK02], BITPOS (BITstream POSitioner) [KJdlTR05] et pBITPOS [KdlTRJ06] permettent les manipulations de configurations partielles des FPGAs de Xilinx, et alors il est possible d'extraire et de déplacer des modules dynamiques, mais il reste impossible d'en produire.

Cette dépendance empêche la production à la volée d'une configuration, et donc la spécialisation d'un module. Nous avons pu cependant contourner cette limite en développant une bibliothèque qui fournit une API qui permet de produire des configurations du FPGA. Les détails sur la technique utilisée pour retrouver l'information manquante et les détails sur l'API de la bibliothèque construite à partir de cette information sont décrits dans le Chapitre 5.

Le manque de documentation du format du bitstream des FPGAs de Xilinx fut un obstacle majeur qui limita le progrès de la compilation dynamique vers ces FPGAs. La bibliothèque que nous avons développée est donc une contribution indirecte importante de notre travail.

3.4 Conclusion

La méthodologie utilisée pour démontrer la viabilité de la compilation dynamique d'applications pour des FGPAs configurables consiste à développer un système complet capable d'enrichir son architecture par des instructions produites à la volée et spécialisées selon des paramètres dynamiques de l'application. Les applications sont écrites dans un langage de haut niveau et sont exécutées sur le système Gambit-C : un environnement d'exécution complet permettant la compilation et l'interprétation de code Scheme. Une portion du code de l'application est synthétisée à la volée pour un FPGA disponible sur le marché.

L'évaluation du système s'effectue par l'exécution d'applications non triviales qui modifient dynamiquement certaines sections critiques pour améliorer leurs temps d'exécution.

L'aboutissement de notre recherche démontre que ce type d'exécution mérite plus d'implication de la part du domaine académique et de l'industrie. Avec cette technologie enfin à la portée de chacun, nous pensons que les applications qui pourraient tirer profit de l'accélération matérielle seront de plus en plus nombreuses.

Chapitre 4

Architecture matérielle

Pour supporter l'exécution reconfigurable et la production à la volée de modules, nous avons développé une architecture matérielle reconfigurable. Le coeur de cette architecture est un FPGA VirtexII-Pro de Xilinx qui permet la reconfiguration dynamique et partielle. Le support de la reconfiguration dynamique des FPGAs permet l'ajout et le retrait à la volée de modules matériels, ce qui est requis pour le type d'exécution que nous proposons. Cette nouvelle fonctionnalité ajoutée par Xilinx fut l'élément déclencheur de nos travaux de recherche vers la synthèse à la volée puisque nous observions que tous les mécanismes matériels requis étaient enfin disponibles dans une même puce.

Dans ce chapitre, nous détaillons les étapes de la conception de l'architecture. Vu le caractère innovateur de notre approche, de nombreuses complications liées aux outils actuels ont dû être contournées. Nous les mentionnons au passage dans les sections appropriées.

4.1 Circuit

Le circuit a été produit avec l'outil XPS (Xilinx Platform Studio) de Xilinx. Cet outil permet de construire graphiquement un circuit en combinant des modules fournis par une bibliothèque (port sériel, contrôleur mémoire, ...). La logique produite par l'outil est ensuite synthétisée par les outils de développement de Xilinx (`ngdbuild`, `map`, `par` et `bitgen`) pour obtenir la configuration initiale.

La Figure 4.1 contient les différents modules de notre système. Les boîtes grises pâles sont implantées avec les blocs configurables (CLB) tandis que les boîtes foncées sont des primitives intégrées.

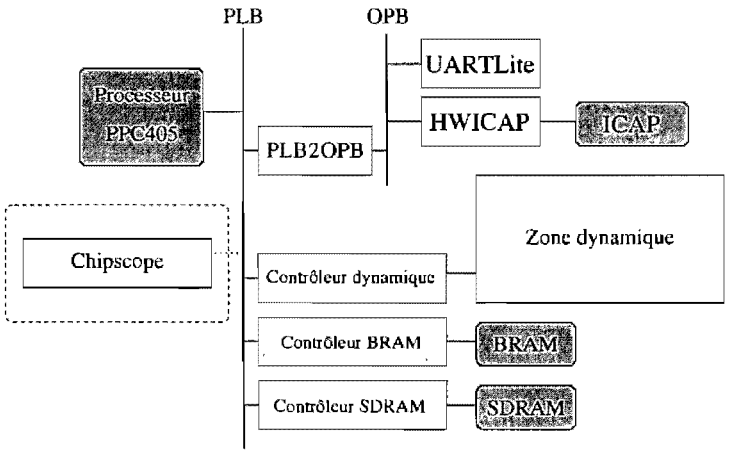


FIG. 4.1 – Design matériel de l'architecture Dynabit

Le processeur PPC405 est au coeur du système et son bus, le PLB (*Processor Local Bus*) permet la communication avec les modules. Ceux nécessitant un couplage proche sont connectés au bus PLB tandis que les autres sont connectés au bus OPB (*On-Chip Peripheral Bus*). La communication avec l'usager s'effectue au travers du port sériel. De plus, deux types de mémoire sont disponibles : la mémoire interne au FPGA (bloc BRAM, 64 Ko) et la mémoire externe SDRAM (256 Mo). La reconfiguration se fait via le HWICAP qui possède une interface de communication de plus haut niveau que l'ICAP. Une composante optionnelle mais présente dans notre architecture est le module chipscope qui permet d'observer les transferts de données sur le bus, ce qui s'avère utile pour déboguer le système.

4.1.1 Contraintes physiques

Des contraintes spécifiques s'appliquent aux applications dynamiques sur le VirtexII-Pro afin d'éviter des bris matériels et des mauvais fonctionnements. Ces contraintes sont documentées dans le manuel de Xilinx qui explique les étapes de développement d'applications dynamiques [Xil02].

Un module reconfigurable doit :

- utiliser la hauteur complète du FPGA (colonne entière),
- posséder une largeur d'au moins quatre tranches (*slices*),
- être un multiple de quatre tranches,
- inclure les IOBs immédiatement au-dessus et en-dessous de la zone reconfigurable,
- communiquer au travers de *Bus Macro*.

Pour parvenir à respecter ces contraintes, il est possible de spécifier aux outils de placement des contraintes de placement via un fichier UCF (*User Constraints File*). Il est malheureusement impossible d'en spécifier pour le routage, ce qui a comme effet que certaines interconnexions requises par le design statique passent par la zone configurable dynamiquement. En conséquence, l'outil *Dynaroute* a été développé afin de corriger les routes problématiques (voir Section 4.1.2).

Après avoir étudié le format du *bitstream*, nous avons remarqué que les contraintes ne sont pas liées à des limites physiques et que la plupart proviennent en fait de la position des bits de configuration et du mécanisme de configuration par tranche. La tranche est la plus petite unité de configuration adressable et couvre la hauteur complète du FPGA, ce qui explique la première contrainte. Puisque l'écriture des bits de configuration ne peut causer d'impulsion transitoire (*glitch-less*), il est possible de ne modifier qu'une partie d'une tranche sans affecter le système en cours d'exécution, à l'exception des tranches configurant des registres. Par conséquent, les IOBs et les CLBs présents dans une même colonne que la zone dynamique ne peuvent être utilisés qu'à la condition que les registres ne le soient pas ou qu'il soit acceptable que leur état puisse être perdu lors d'une reconfiguration.

L'utilisation de *Bus Macro* est nécessaire pour forcer une interface compatible (e.g. les mêmes fils) entre les différents modules dynamiques qui sont généralement compilés séparément. Nous proposons une meilleure solution que celle de Xilinx en permettant des bus de communication plus larges (voir Section 4.1.3).

4.1.2 Dynaroute

Les outils de Xilinx ont été conçus pour le développement d'applications statiques, mais la reconfiguration dynamique nécessiterait des modifications pour, par exemple, forcer le passage d'une route par certains fils. Actuellement, il est possible d'imposer le positionnement des primi-

tives sur des sites précis mais il est encore impossible de forcer le routage via des fils spécifiques. Comme il serait utile de pouvoir garantir ou proscrire le positionnement de fils, les outils de Xilinx ne répondent pas complètement aux besoins de la communauté.

En réponse à ce problème, nous avons conçu le logiciel Dynaroute qui permet de modifier un design afin de préciser le positionnement des fils. Au lieu de réécrire un algorithme de routage, nous utilisons les outils standards de synthèse et corrigeons ensuite les problèmes avec nos outils.

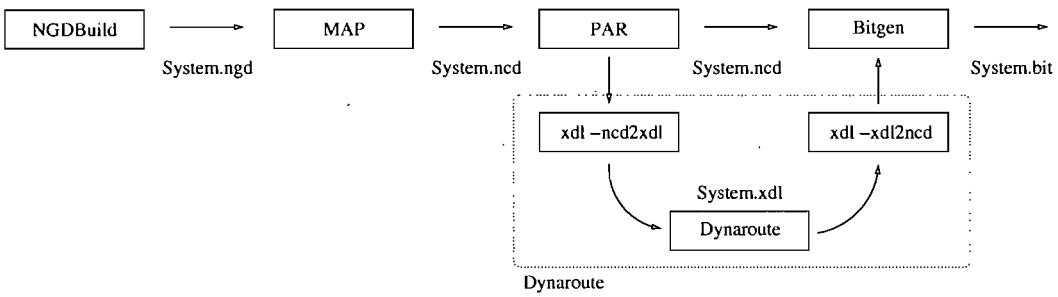


FIG. 4.2 – Intégration de Dynaroute dans la synthèse

La Figure 4.2 montre comment intégrer l'outil Dynaroute au cycle de développement de Xilinx. Le développeur effectue la synthèse de haut-niveau (`NGDBuild`), la correspondance technologique (`map`), le placement et le routage (`par`). Avant de produire le bitstream, le développeur utilise l'outil XDL pour obtenir la description du design sous format XDL (*Xilinx Description Language*), puis Dynaroute modifie le routage et produit le design modifié sous le même format. Par la suite, l'outil XDL convertit à nouveau le design sous le format natif (NCD, *Natif Circuit Description*) et le processus de synthèse peut être complété. Le design ainsi obtenu respecte les contraintes supplémentaires requises par l'architecture.

Dynaroute permet de corriger certaines lacunes actuelles des outils de Xilinx. Il s'applique par exemple au développement d'applications où les contraintes de temps ne sont pas respectées, dû à une latence trop élevée de certaines routes, qui nécessite une intervention humaine pour appliquer des correctifs.

```
(edit-design
  "implementation/system.xdl"      ;; input
  "implementation/system-out.xdl" ;; output
  (lambda ()
    (forbid-zone 'SLICE_X44Y150 'SLICE_X88Y134)
      (let ((nets-to-reroute
              (filter-strings "plb_dynamic_zone/.*/dpipe/d.*"
                              (nets-in-forbidden-zone))))
        (reroute-nets nets-to-reroute))))
```

FIG. 4.3 – Exemple de script pour Dynaroute

La Figure 4.3 contient un script utilisé par Dynaroute qui interdit aux fils de passer par la zone (SLICE_X44Y150 à SLICE_X88Y134), la zone dynamique de notre système.

4.1.3 Bus Macro

Les communications entre les zones statiques et dynamiques doivent respecter des contraintes pour garantir un bon fonctionnement du système. Selon les propositions de Xilinx, toute route doit traverser les frontières des zones au travers d'un *bus macro* [Xil02] pour garantir une interface compatible entre les différents modules dynamiques chargés. La Figure 4.4 contient un exemple de la macro proposée par Xilinx.

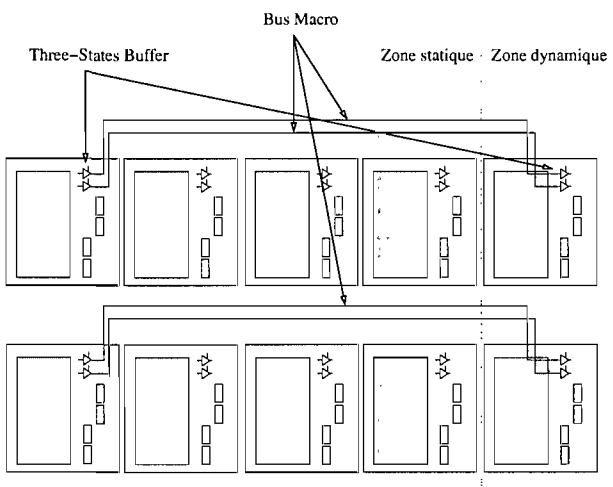


FIG. 4.4 – Bus Macro proposé par Xilinx

L'idée derrière cette technique est d'utiliser les *three-states buffers* présents dans les CLBs.

Puisque le développeur peut forcer le positionnement de primitives et qu'il n'existe qu'un chemin entre deux de ces primitives, toutes les instances d'un module dynamique qui les positionnent aux mêmes endroits sont garanties d'avoir une interface identique. Cette technique a été conçue pour passer outre les limites actuelles des outils qui n'ont pas le concept de contraintes sur les routes. Notons que cette technique force une distance de 4 CLBs entre les macros et limite les communications à 2 bits par CLB.

Cette macro n'est pas utilisée par notre système puisqu'elle impose des contraintes trop importantes. Dynaroute résout le problème corrigé par le *Bus Macro* avec plus de latitude en permettant de spécifier des contraintes plus évoluées sur le routage, ce qui permet de développer des macros plus complexes.

Pour produire une macro de communication, nous effectuons la synthèse de notre circuit avec les outils de Xilinx et puis corrigeons certaines routes avec Dynaroute. Des signaux pour communiquer avec la zone dynamique ont en plus été ajoutés à notre design. Ces signaux traversent deux LUTs que nous positionnons de part et d'autre de la frontière, ce qui incite l'outil de Xilinx à faire un routage qui respecte quasiment nos caractéristiques requises, plaçant les routes dans une région localisée à la frontière. Dynaroute modifie alors les routes qui traversent la frontière pour respecter l'interface que nous avons pré-établie. Lors de l'exécution, les LUTs et les routes positionnées dans la zone dynamique sont simplement ignorées et sont écrasées par la reconfiguration. L'interface est figée et il est possible de s'y connecter dynamiquement.

La Figure 4.5 contient un exemple de la construction d'une macro avec notre technique. La première rangée de CLBs est produite par les outils de Xilinx puis corrigée par Dynaroute. Dans notre cas, nous utilisons les fils statiques doubles et hex. Les fils statiques ont comme origine et comme destination deux matrices de routage différentes et ne possèdent aucun point de configuration autre que la source et la destination. Les fils doubles ont une distance de deux CLBs tandis que les fils hex ont une distance de six CLBs. La deuxième rangée de la Figure 4.5 est l'ensemble réellement conservé de la macro. La zone de gauche est la zone statique et n'est jamais modifiée tandis que la zone de droite est modifiée dynamiquement et effectue les communications au travers de l'interface.

Il y a des avantages évidents à utiliser ce type de macro. Par exemple, il est impossible d'avoir plusieurs sources pour un fil car les fils utilisés sont alimentés seulement par une seule matrice de routage, ce qui évite les conflits entre les zones statiques et dynamiques. De plus, il

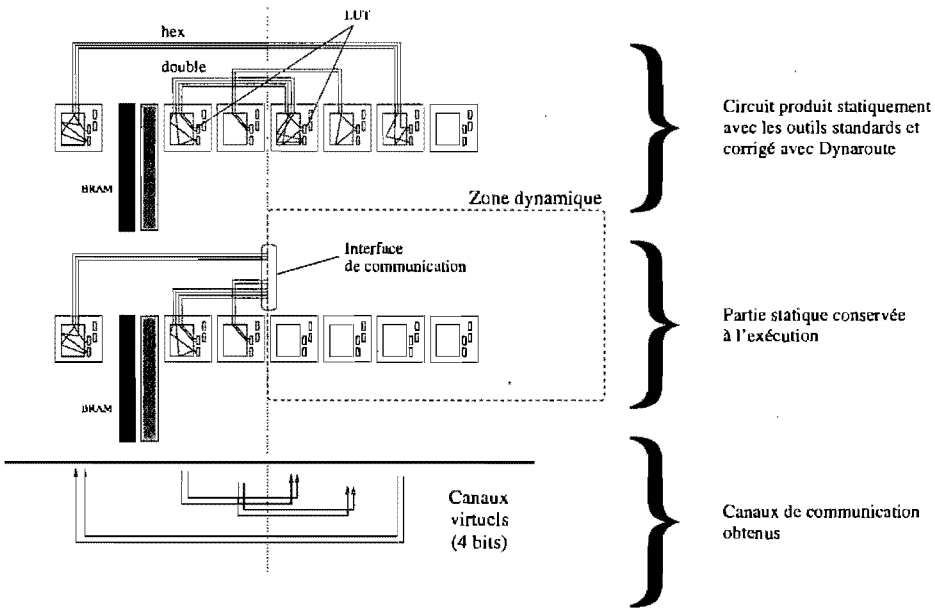


FIG. 4.5 – Bus Macro utilisé par notre système

est possible de créer des bus de communication plus larges que deux bits par CLB.

Pour notre architecture, nous avons construit une macro qui possède quatre bus d'entrée et deux bus de sortie et où chaque CLB reçoit ou émet 8 bits. La troisième rangée de la Figure 4.5 contient les canaux virtuels que notre système utilise. En dupliquant cette macro, nous avons construit des canaux virtuels de 16 ou 32 bits.

4.1.4 Connexion au processeur

Notre système a une interface entre le processeur et la zone dynamique par laquelle l'application peut communiquer avec les instructions créées dynamiquement. Le couplage entre la zone dynamique et le processeur influence considérablement les performances. Le PowerPC permet différents niveaux de couplage et nous avons envisagé les possibilités suivantes : l'APU (*Auxiliary Processing Unit*), le PLB (*Processor Local Bus*) et OCM (*On-Chip Memory*).

L'APU était le meilleur choix, mais malheureusement une erreur de conception des Virtex-II rend impossible son utilisation, comme Xilinx l'a confirmé.

L'OCM est une interface mémoire qui permet d'ajouter au processeur des blocs de mémoire

avec la même latence que l'antémémoire, tandis que le PLB est un bus qui respecte le protocole CoreConnect d'IBM. Comme ces deux alternatives sont intéressantes, nous avons décidé d'implanter notre protocole sur une interface mémoire. Ainsi notre interface peut être utilisée tant sur le PLB que sur l'OCM en connectant le contrôleur de mémoire approprié. Notre interface possède un bus de données, un bus d'adresse et des signaux de contrôle ; cette abstraction permet de nous connecter sur le bus OPB (*On-Chip Peripheral Bus*) et ainsi de nous connecter aussi à un processeur synthétisé (*soft processor*) tel que le MicroBlaze.

Un avantage notable de cette abstraction est que le contrôleur de mémoire, déjà implanté, supporte le protocole du bus et notre système ne reçoit essentiellement que les données et les signaux de contrôle, ce qui constitue une version simplifiée du protocole. De plus, elle nous permet d'utiliser le mode de communication en rafale (*Burst Mode*) de la mémoire.

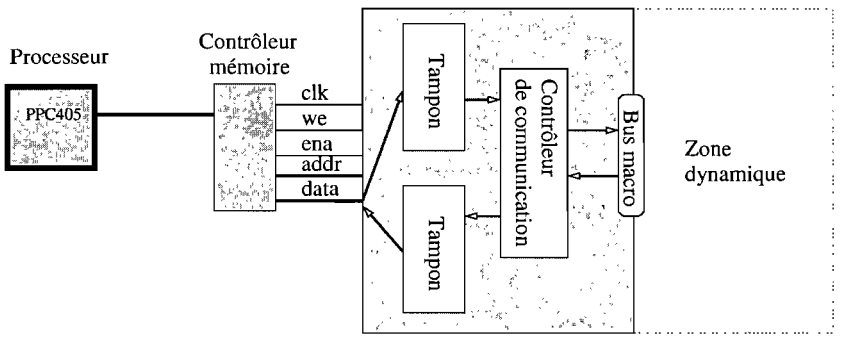


FIG. 4.6 – Contrôleur dynamique (*pipeline*)

La Figure 4.6 représente le contrôleur de la zone dynamique que nous avons implanté en utilisant l'abstraction proposée pour se connecter au PLB. Pour émettre une donnée dans la zone dynamique, l'application effectue une écriture mémoire de la donnée à une adresse fixe, puis le processeur envoie la ligne de l'antémémoire correspondante au contrôleur de mémoire. La ligne de l'antémémoire (8 entiers de 32 bits) est accumulée sur plusieurs cycles dans le tampon d'entrée. Lors de la prochaine transaction de ligne de l'antémémoire, ce tampon est envoyé au contrôleur de la zone dynamique qui communique les données à l'instruction spécialisée. Le rôle de ce contrôleur est de positionner les paramètres sur les bonnes entrées du *Bus Macro* selon le nombre d'opérandes de l'instruction présente. Les informations telles que le nombre d'opérandes sont encodées dans certains bits d'adresse, ce qui permet de se passer de communication avec le contrôleur. Les résultats sont ensuite placés dans un tampon de sortie qui est accessible via une

plage d'adresses.

Le mode de fonctionnement de notre mécanisme de communication est adapté aux calculs qui s'effectuent en *pipeline*. D'autres interfaces de communication seraient mieux adaptées pour des applications possédant des dépendances de données et de contrôle, mais notre choix de conception est un compromis entre la complexité de développement et la performance. Comme nous voulons consacrer notre temps sur les techniques de compilation rapide, ce mode de communication nous permet des tests réels sans pour autant être trop complexes à implanter.

4.2 Résultats

La version finale du système a été développée sur les outils XPS 9.1 et ISE 9.1. Le système a été construit spécifiquement pour la carte de prototypage ML310 de Xilinx.

La fréquence maximale imposée à la logique configurable *(CLB)* par l'outil de développement XPS de Xilinx pour la carte ML310 est de 100 MHz. Ainsi, le système résalisé est cadencé à une fréquence d'horloge de 100 MHz. La qualité du FPGA sur la carte ML310 (e.g. *grade -5*) impose une fréquence limite de 350 MHz pour le processeur. Afin de réduire la complexité de notre système, les zones dynamiques sont cadencées avec la même horloge que le système statique, soit 100 MHz.

La Figure 4.7 contient une capture d'écran du système réalisé. Le système contient deux zones dynamiques (haut et bas) représentées par les boîtes grises pâles. Chaque zone dynamique contient 960 CLBs (3840 tranches). Le port d'accès se trouve en haut de chaque zone dynamique.

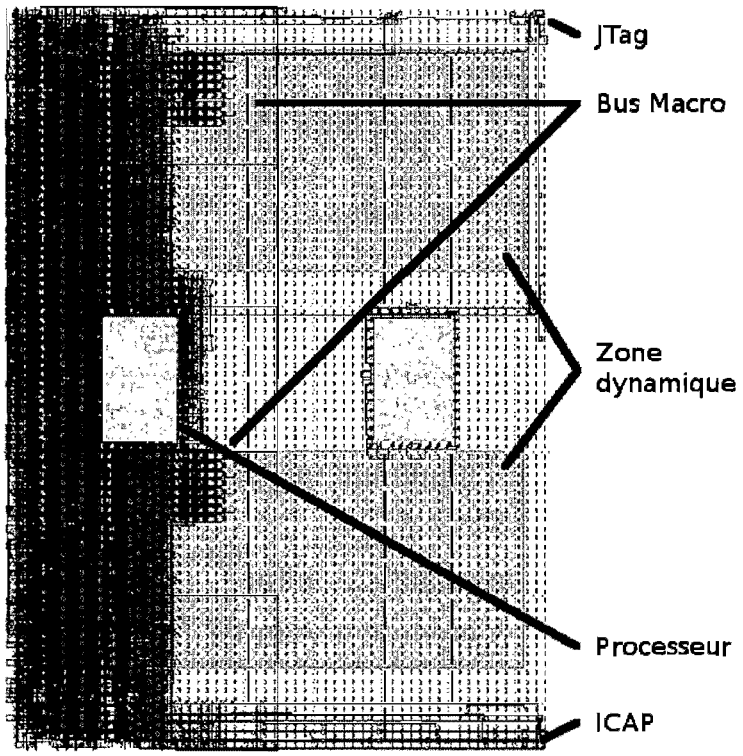


FIG. 4.7 – Capture d'écran du système réalisé

4.3 Conclusion

Les problèmes de conception matérielle rencontrés par le développement de ce type d'architecture ont pu être contournés. Les limitations ne sont pas physiques et découlent de la nouveauté de la reconfiguration dynamique et de sa mauvaise intégration dans les outils actuels.

Nous avons complété un système fonctionnel et stable. Ce système contient deux zones dynamiques sans aucun conflit de fils et possède des ports de communication qui relient la zone statique et dynamique. Ainsi, la communication entre le processeur et la zone dynamique peut s'effectuer au travers d'un contrôleur avec un temps de latence équivalent à un accès mémoire.

Enfin, ce système peut être utilisé par d'autres applications dynamiques, ce qui le rend encore plus attrayant.

Chapitre 5

Bibliothèque de reconfiguration dynamique

Le comportement des composantes du FPGA est contrôlé par des bits de configuration conservés dans une mémoire volatile. La reconfiguration dynamique d'une partie du FPGA s'effectue par la modification de ces bits accessibles au travers de l'ICAP (*Integrated Communication Access Port*). Ainsi, l'échange de modules dynamiques consiste à charger un bloc de bits dans cette mémoire de configuration. Cependant la mémoire de configuration n'est pas documentée, ce qui a comme conséquence que la production dynamique de configurations est hors de portée même pour les experts.

Avec l'outil XDL de Xilinx, il est possible d'obtenir un rapport détaillé et presque complet contenant la structure du FPGA ainsi que ses points de configuration et leurs domaines. La structure du FPGA y est clairement détaillée et l'information nécessaire à la construction de notre système y est présente, excepté l'encodage des points de configuration. Par analogie, le problème rencontré est similaire à celui d'un assembleur où le jeu d'instructions est connu mais pas l'encodage des instructions. Comme nous possédons un générateur de bitstream, il est possible de retrouver l'encodage des points avec une technique naïve qui consiste à générer une configuration pour chaque valeur de chaque point configurable et à déterminer quels sont les bits affectés dans la configuration. Cependant, le nombre de points programmables est trop élevé pour utiliser une telle technique. Nous avons donc utilisé une technique algorithmiquement

plus efficace. Le premier article de ce chapitre, soumis à la revue *Journal ACM Transactions on Reconfigurable Technology and Systems*, démontre la technique utilisée et justifie le besoin pour cette information.

Une fois l'encodage déterminé, la prochaine étape à franchir consiste à réaliser un générateur de configuration à la volée. Notre première approche a été d'utiliser un ensemble de tuiles abstraites qui sont des circuits effectuant des opérations de base (e.g. addition, décalage, ...). L'idée est de produire manuellement de petites tuiles et d'utiliser la connaissance du format du bitstream pour permettre des interfaces compatibles entre les tuiles de sorte que la sortie d'une tuile soit facile à connecter à l'entrée d'une autre tuile. Cela évite l'utilisation de *bus macro* et permet aux tuiles d'être superposées. Grâce à cette approche nous avons pu réaliser un premier JIT qui ne dépendait pas des outils de Xilinx. Ce JIT est décrit dans le deuxième article, présenté à la conférence *NEWCAS : International Northeast Workshop on Circuits and Systems* en 2007 [BFD07].

Pour faciliter la recherche dans ce domaine, nous avons senti le besoin d'une bibliothèque et d'une suite d'outils adaptés à la reconfiguration dynamique. L'existence de cette bibliothèque au début de notre projet aurait été un tremplin nous permettant d'attaquer dès le début les problèmes reliés à la synthèse rapide. Le troisième article, publié à la conférence *NEWCAS : IEEE Northeast Workshop on Circuits and Systems* en 2008 [BDFD08], contient un exemple d'une application qui utilise notre bibliothèque et démontre ainsi que des applications autres que la synthèse dynamique peuvent tirer profit de notre bibliothèque et de nos outils.

Logarithmic Time FPGA Bitstream Analysis : a Step Towards JIT Hardware Compilation

Etienne Bergeron, Louis-David Perron, Marc Feeley, Jean Pierre David

Abstract— **Just-in-time (JIT) compilation is frequently used in software engineering to accelerate program execution. Parts of the code are translated to machine code at run time to speedup their execution by exploiting local and dynamic information of the computation. Modern FPGAs manufactured by Xilinx allow partial and dynamic configuration. Such features make them eligible platforms for JIT hardware compilation. Nevertheless, this has not been achieved until now because the mapping between a bitstream and the programmable points inside these FPGAs is not documented. In this paper, we propose a methodology to retrieve the relevant information in logarithmic time per bit by methodically using the tools distributed by Xilinx. We give a practical case study which details the analysis of a Virtex-II Pro FPGA bitstream. The mapping of CLBs, BRAMs and multipliers has been fully determined and preliminary results demonstrate that a processor embedded in an FPGA can compile, place and route arithmetic and logic expressions inside the FPGA within a few milliseconds.**

I. Introduction

In the Von Neumann architecture the behavior of a generic device, the processor, is partially defined by the internal architecture of the processor and partially defined by a program stored in a memory. To speed up execution of a computation a processor can generate in memory the specialized machine code it will run immediately afterwards. Dynamic code generation is now commonly used in the efficient implementation of virtual machines for programming languages. It is an integral part of the just-in-time (JIT) compilation technique [BDB00]. HotSpot [Sun99], for example, uses a JIT compiler to dynamically translate Java bytecode into optimized machine code thus bypassing the relatively slow interpretation process.

FPGA technology also uses a memory to determine a circuit's behavior, but at a much lower level of abstraction. Most FPGAs are configured at power up time by downloading a bitstream in their distributed configuration memory. The bitstream encodes the set of programmable points that determine the configuration of the FPGA. A programmable point affects the local behavior of a small sub-circuit such as a Look Up Table (LUT), multiplexer, routing logic, and dedicated circuit. Typically each programmable point is encoded with a small number of bits in the bitstream. The bitstream thus entirely defines the FPGA's behavior and can be seen as a binary representation of a complex digital circuit expressed at a level of abstraction close to the gate. Recent FPGAs manufactured by Xilinx also support partial and dynamic configuration. A part of the circuit implemented in the FPGA can be modified at run time while the rest of the circuit is in operation. The running part can reconfigure the other part through the embedded configuration port (ICAP). Such concept is known as self-reconfiguration and has been formally defined in 2002 by Sidhu and Prasanna[SP02]. The present work only addresses a subset of self-configurating devices : the FPGA. In this context, the *metacomputation* concept defined in [SP02] is actually JIT *Hardware Compilation (HC)*, where HC means the combination of several steps among the following:

- Generation of a digital circuit at a given abstraction level.
- Synthesis.
- Technology mapping.
- Optimization (delay/area).
- Place and route.
- Bitstream generation (mandatory).
- Configuration (mandatory).

A major hurdle however is that HC is a long and complex task achieved by proprietary tools and often requiring large amounts of memory. A full compilation may take several hours and Gigabytes of memory. A simplistic approach is to compile before run time a set of possible partial configurations and to dynamically switch between them at run time. This is the design flow recommended by Xilinx and all research projects involving dynamic configuration for recent FPGAs rely on it. In common practice there is no alternative because the mapping between the bitstream and the programmable points in the FPGA is not documented. The use of Xilinx development flow and tools seems mandatory. Nevertheless, some applications require actual JIT HC because it is not possible to compile all the possible configurations before run time. Typical applications are cryptography (too many keys, plaintexts or ciphertexts), neural networks (too many topologies and/or coefficients), pattern matching (when patterns are known at runtime only), generic code accelerator (must apply to any application, not known at runtime) etc.

In this paper, we propose a way to analyze an FPGA bitstream to find the mapping between a large subset of the programmable points and their associated bits in the bitstream. We have focused on the programmable points related to logic blocks and routing (CLBs), memory blocks (BRAMs) and multipliers because these are the only resources required to implemented JIT HC.

Finding the best way to implement JIT HC will require much research. The *Warp* project [LSV06] already rele interesting results for a custom made FPGA. In this paper, w do not address this issue. Our work concentrates on the efficient and automatic analysis of commercial FPGA bitstreams. Our methodology has a logarithmic complexity per bit in the

bitstream and does not make any assumption on the regularity of the FPGA's structure. The methodology has proved to be fully functional for a Virtex II Pro FPGA. Furthermore work in progress in the field of JIT HC demonstrates that the cited FPGA can compile, place and route arithmetic and logic expressions autonomously within a few milliseconds.

The paper is organized as follows: Section II is dedicated to the related work in the field of FPGA bitstream manipulation and reverse engineering. Section III and IV present the formalism and the theoretical aspects of our methodology to analyze an FPGA bitstream. Section V proposes an application of this methodology to a Xilinx Virtex-II Pro FPGA. Section VI presents the results. Some work in progress in applications and tools demonstrate that our approach is fully functional and promising in the field of JIT HC. They are proposed in Section VII. Section VIII concludes this work.

II. Related work

JBits [GLS99] is a tool developed by Xilinx to handle dynamic reconfiguration. It provide a Java API to generate bitstreams for partial and dynamic reconfigurations. Version 2.8 of JBits also includes a simulator, but it is no longer available in more recent versions. Furthermore, JBits is limited to a restricted set of FGPAs (e.g. JBits 3 only supports Virtex-II). Many projects that generate or manipulate bitstreams use the JBits API to abstract the bitstream manipulation. For instance, JHDLBits [PHP+04], [PHPA04] is a language bridge between JHDL [BH98] and JBits that allows applications running in JHDL to handle dynamic reconfiguration. JPG [RS02] is another tool that uses JBits and the Xilinx description language (XDL) representation to generate partial bitstreams.

JBits has been used to attempt to reverse engineer a bitstream. The VirtexTools project [Fra03] tried to use this approach to reverse engineer the format of Spartan-II XC2S00 bitstreams in order to develop freeware tools for creating and manipulating such bitstreams. Unfortunately the tools have limited functionality and the project was abandoned in 2003. XPART, also developed by Xilinx, is a similar project except that it does not require a Java framework. Unfortunately, it has been abandoned too.

Another approach to reverse engineer a bitstream is to reverse engineer proprietary tools or intermediate files. For instance, ADB [Ste02] uses the BitFile description (BFD) files, which describe the bitstream structure. But the format of this file is proprietary to Xilinx and undocumented.

Some projects just need to manipulate bitstreams in a way that doesn't require knowledge of its format. For instance, Parbit [HL01] uses relative addressing to manipulate *opaque components* (modules) through the regular structure of an FPGA. Such projects would benefit from documentation of the bitstream format.

It is also possible to manually reverse engineer the bitstream format by using FPGA editor or Bitgen tools [RW07]. Such approach relies on the regular structure of the FPGA and cannot be automated. Each device family requires a manual investigation, which is a tedious and time consuming job.

Debit [Not07] is a recently announced open-source project which provides a tool for converting Xilinx bitstreams back into the XDL representation. This tool supports several devices in the Virtex family, but not the Virtex-II Pro. The methodology used for reverse engineering the Xilinx format relies on several assumptions referred as "Coherency hypothesis" and "Morphism hypothesis" [NER08]. These assumptions require that the structure of the FPGA be regular and known. Furthermore, each configuration block must be encoded by the same pattern in the bitstream. Given these assumptions, the methodology compares the encoding of each configuration block with their associated bits in the bitstream and deduces the mapping.

Our methodology, which was outlined in a previous work [BFD07], does not make any assumption on the regularity of the FPGA structure but our test case on the Virtex-II Pro confirms the regularity, with a few exceptions.

III. Formalism

An FPGA is a configurable digital circuit. Its behavior can be tailored to a specific application through the process of *configuration* which is performed when the FPGA is powered-up and may occur during execution in the case of dynamically reconfigurable FPGAs.

Conceptually, the configuration of a given FPGA is a vector of *programmable point* settings, $P = \langle P_0, P_1, ..., P_{|P|-1} \rangle$, where $|P|$ is the number of programmable points. P_i, the setting of the programmable point i, determines the behavior of a specific part of the whole FPGA. Each programmable point is constrained by the FPGA architecture to a specific domain of discrete values, $P_i \in D_i$.

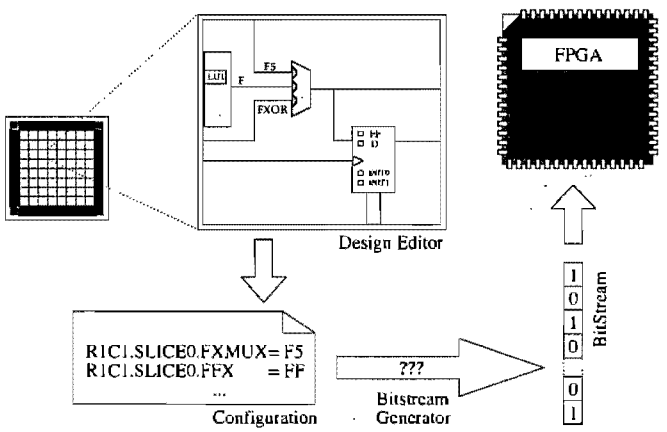


Fig. 1. Typical design flow from design description, to programmable point settings, to bitstream

For user convenience, development environments and documentation define symbolic names to identify the programmable points and the set of possible values. For example, on the Xilinx VP2, R1C1.SLICE0.FXMUX $\in$ {F5, FXOR, F}. Without loss of generality, we will identify the programmable points and their domains numerically, i.e. $D_i = \{0, 1, ..., |D_i| - 1\}$ where $|D_i|$ is the cardinality of the domain D_i. For example, if R1C1.SLICE0.FXMUX is the programmable point P_{183} then $|D_{183}| = 3$, $D_{183} = \{0, 1, 2\}$, and the following encoding is used: $0 \Rightarrow$ F5, $1 \Rightarrow$ FXOR, and $2 \Rightarrow$ F.

In addition to the domain vector $D = \langle D_0, D_1, ..., D_{|P|-1} \rangle$, the FPGA architecture defines a set of constraints between programmable points. Each constraint restricts the domain of a programmable point as a function of the setting of other programmable points. For example, in a Xilinx VP2 slice, the configuration domain of the flip-flops, normally {FF, LATCH, OFF}, is restricted to {FF, OFF} when the reset type is set to SYNC. A *consistent configuration* is a configuration that respects all FPGA constraints.

A bitstream is a vector $B = \langle B_0, B_1, ..., B_{|B|-1} \rangle$ of bits ($B_j \in \{0, 1\}$) encoding the configuration P which is decoded during the FPGA configuration process. Obviously the encoding and decoding techniques used must match. Various encoding techniques are currently available for FPGAs, including compressed and encrypted bitstreams.

A common encoding that simplifies the configuration process is the *plain bitstream*. It uses a fixed-length bitstream, assigns to each P_i a group of possibly nonadjacent bits in the bitstream that encode the setting of P_i, and assigns to each element of D_i a distinct encoding bit pattern for this group of bits. More precisely the *address set* $A_i = \{A_{i,0}, A_{i,1}, ..., A_{i,|A_i|-1}\}$ indicates the set of bit positions in the bitstream that constitute the group of bits encoding P_i.

$C_{i,v}$ is a bit vector of length $|A_i|$ which indicates the value of the bits in the group that codes P_i for the setting $P_i = v$. The first bit of $C_{i,v}$ is the value of the bit whose position in B is the lowest address in A_i; the second bit of $C_{i,v}$ corresponds to the next lowest address in A_i; and so on.

For example, if R1C1.SLICE0.FXMUX is the programmable point P_{183} then possibly $A_{183} = \{48678, 48734\}$, and $C_{183,0} = \langle 0, 1 \rangle$, $C_{183,1} = \langle 1, 0 \rangle$, and $C_{183,2} = \langle 0, 0 \rangle$. This would mean that the encoding of the programmable point setting R1C1.SLICE0.FXMUX=F5 requires setting $B_{48678} = 0$ and $B_{48734} = 1$.

Because of the architectural constraints between programmable points, some bits in the bitstream may be shared by the encoding of multiple programmable points. Moreover, some bits may not be directly related to the encoding of programmable points (constant bits such as framing bits, device identification bits, ...). Some bits may have an arbitrary value (time stamp, serial number, ...), or they may be computed from other bits in the bitstream (checksums). Finally, some programmable points may not be related to any bit in the bitstream, when $|D_i| = 1$ or when the programmable point has a purely advisory purpose.

The problem we aim to solve is to determine for each programmable point P_i, the address set A_i and the encoding bit patterns $C_{i,0}$, $C_{i,1}$, ..., $C_{i,|D_i|-1}$. We also aim to determine which bits of the bitstream are constant, arbitrary, and computed. To achieve this goal, we suppose that we have access to a tool capable of generating a plain bitstream from a higher level description of the programmable points, which is typically the case. In the following sections, we will call this tool the *BitStreamGenerator*.

IV. Relations and Logarithmic Mapping

We will make some simplifying assumptions and then gradually remove the simplifications to handle the general case. We assume that all programmable points have only two possible values, X and Y, that there are no constraints between programmable points, and that every bit in the bitstream is p of the encoding of a programmable point (i.e. there are no constant bits, time stamps, etc). In this context all assignments of X and Y to programmable points is a consistent configuration. It is still the case that several bits in the bitstream can be used to encode a given programmable point.

We define the relation $\mathcal{I} \rightarrow_v \mathcal{J}$, where v is a programmable point value, as

$$\{i \mid P_i = v\} \rightarrow_v \{j \mid B_j = 1\}$$

This relation maps the set of P's programmable points whose settings have the value v to the set of bit positions in P's bitstream that are equal to 1.

The intersection and complement of such relations are defined using the set intersection and set complement operators:

$$(\mathcal{I} \rightarrow_v \mathcal{J}) \cap (\mathcal{I}' \rightarrow_v \mathcal{J}') = (\mathcal{I} \cap \mathcal{I}') \rightarrow_v (\mathcal{J} \cap \mathcal{J}')$$
$$\overline{\mathcal{I} \rightarrow_v \mathcal{J}} = \overline{\mathcal{I}} \rightarrow_v \overline{\mathcal{J}}$$

Given a set of relations $R = \{R_0, R_1, ..., R_{|R|-1}\}$, a programmable point P_i is *isolated* within a set $R' \subseteq R$ when the relation $\bigcap R'$ has the singleton set $\{i\}$ as domain, i.e. $\bigcap R' = (\{i\} \rightarrow_v \{...\})$. This arises when

$$\forall (\mathcal{I} \rightarrow_v \mathcal{J}) \in R', i \in \mathcal{I}$$

and

$$\forall i' \neq i, \exists (\mathcal{I} \rightarrow_v \mathcal{J}) \in R' \text{ s.t. } i' \notin \mathcal{I}$$

The set R *fully isolates* P when for all P_i there exists a subset of R that can isolate P_i.

A. Simple case

We will further assume that the programmable point value X is encoded with one or more 0 bits and the programmable point value Y is encoded with one or more 1 bits. Consequently, in the simple case domains contain only two values ($\forall i, |D_i| = 2$), there are no constraints between programmable points, there are no constant, arbitrary or computed bits, and relations $\{\} \rightarrow_Y \{\}$ and $\{0, 1, ..., |P| - 1\} \rightarrow_Y \{0, 1, ..., |B| - 1\}$ hold.

From relation $\mathcal{I} \rightarrow_Y \mathcal{J}$ we know that setting $P_i = Y$, where $i \in \mathcal{I}$, causes some of the bit positions in $\mathcal{J}$ to be set to 1, so for each a in A_i we have $a \in \mathcal{J}$ (equivalently $A_i \subseteq \mathcal{J}$). This means that by activating some programmable points we can narrow down the possible positions by looking at the positions in the bitstream that are activated. By using a set of relations that isolate P_i we can fully determine A_i. The exact set is the resulting positions of the intersection of relations, i.e. $\{i\} \rightarrow_Y \mathcal{J} \Rightarrow A_i = \mathcal{J}$. As an example, let's look at the following mapping:

$$P = \langle P_0, P_1, P_2, P_3, P_4, P_5, P_6 \rangle$$
$$A = \langle \{0\}, \{2, 3\}, \{1, 4\}, \{5\}, \{7\}, \{6\}, \{8, 9\} \rangle$$

Consider the set of relations $R = \{R_0, R_1, R_2, R_3\}$.

$$R_0: \{1, 2, 3\} \rightarrow_Y \{1, 2, 3, 4, 5\}$$
$$R_1: \{2, 3, 5\} \rightarrow_Y \{1, 4, 5, 6\}$$
$$R_2: \{2, 4, 6\} \rightarrow_Y \{1, 4, 7, 8, 9\}$$
$$R_3: \{3, 4, 6\} \rightarrow_Y \{5, 7, 8, 9\}$$

We can isolate P_2 with $R_0 \cap R_1 \cap R_2$ and P_3 with $R_0 \cap R_1 \cap R_3$.

$$\{2\} \rightarrow_Y \{1,4\} \quad \Rightarrow \quad A_2 = \{1,4\}$$
$$\{3\} \rightarrow_Y \{5\} \quad \Rightarrow \quad A_3 = \{5\}$$

The question is how to efficiently generate a set R that fully isolates P. A trivial but slow solution is to generate one relation for each programmable point that maps to their corresponding positions, i.e. $\{ \{i\} \rightarrow_Y A_i \mid i \in \{0,1,...,|P|-1\} \}$. It is estimated that this technique would take several years to compute for a large FPGA. We are interested in a minimal set of relations with the same properties. We propose to use a *logarithmic mapping*.

By definition we have $(P_i = X) \Rightarrow (B_j = 0)$ and $(P_i = Y) \Rightarrow (B_j = 1)$, where $j \in A_i$. If we consider any sequence of relations, the sequence of values P_i takes will match the sequence of values that B_j takes, for all $j \in A_i$. These sequences can be seen as vectors of bits which are the binary encoding of integers. We propose to build a sequence of relations such that the sequence of values P_i takes represents the integer i (where X is taken as bit 0 and Y is taken as bit 1). This way the sequence of values B_j takes will encode the integer i. The mapping $j \in A_i$ can then be deduced straightforwardly, without any time-consuming set manipulation. To fully isolate P we need $|R| = \lceil log_2(|P|) \rceil$.

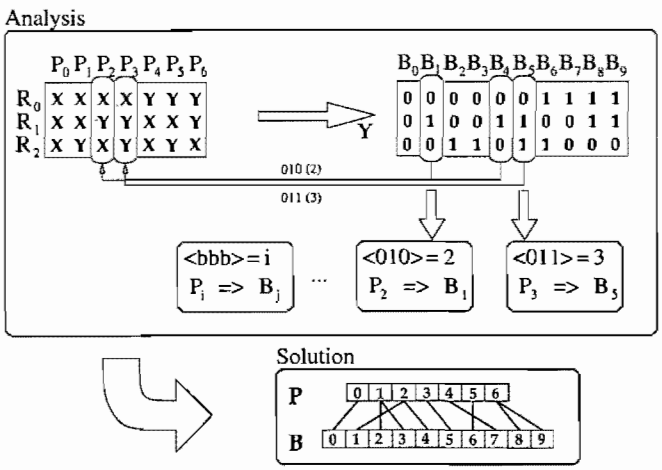


Fig. 2. Simple logarithmic mapping example using 3 relations to determine the mapping when $|P| = 7$ and $|B| = 10$

Figure 2 shows an example of the logarithmic mapping technique. The PV table (sequences of programmable points vectors) is on the left and the BV table (sequences of bitstreams) is on the right. A relation is a row in both tables. Columns are the vectors representing integers. The column P_2 contains the encoded value of 2 ($010 \mapsto XYX$). B_1 and B_4 are the only columns encoding the integer 2. Therefore, we can deduce the mapping $A_2 = \{1,4\}$. This is formalized by Algorithm 1.

B. Handling fixed and negative positions

In the simple case we assumed that both relations $\{\} \rightarrow_Y \{\}$ and $\{0,1,...,|P|-1\} \rightarrow_Y \{0,1,...,|B|-1\}$ hold. All bitstream positions in A_i are toggled by the activation of programmable point P_i and activated bits are always set to 1. A consequence of this property is that we can use the complement of a relation without another bitstream generation because $(\mathcal{I} \rightarrow_X \mathcal{J}) \Leftrightarrow (\overline{\mathcal{I}} \rightarrow_X \overline{\mathcal{J}}) \Leftrightarrow (\mathcal{I} \rightarrow_Y \overline{\mathcal{J}})$.

Algorithm 1: Simple logarithmic mapping

```
1  proc mapping (P, X, Y, A) is
2  begin
3      for k = 0 to ⌈log2(|P|)⌉ − 1 do
4          for i = 0 to |P| − 1 do
5              if bit k of i = 0 then
6                  PV[k][i] ← X
7              else
8                  PV[k][i] ← Y
9              end
10         end
11         BV[k] ← BitStreamGenerator(PV[k])
12     end
13     for j = 0 to |B| − 1 do
14         i ← 0
15         for k = 0 to ⌈log2(|P|)⌉ − 1 do
16             i ← 2 ∗ i + BV[k][j]
17         end
18         A_i ← A_i ∪ {j}
19     end
20 end
```

Typical bitstreams may contain some fixed positions (always the same value) and negative positions ($B_j = 1$ when $P_i = X$ and $B_j = 0$ when $P_i = Y$). The impact on our technique is that fixed positions are either mapped to A_0 or $A_{2^{|R|}-1}$ because the resulting address contains either only 0 bits or only 1 bits. Also, negative positions are mapped to the inverse address. Therefore, modifications to our technique are required.

Opposite relations $\mathcal{I} \rightarrow \mathcal{J}$ and $\overline{\mathcal{I}} \rightarrow \mathcal{J}'$ are used to detect fixed bits. Bits stuck to 1 are in both sets $\mathcal{J}$ and $\mathcal{J}'$. So, $\mathcal{J} \cap \mathcal{J}'$ is the set of bits stuck to 1. Similarly $\overline{\mathcal{J}} \cap \overline{\mathcal{J}'}$ is the set of bits stuck to 0. Therefore, fixed positions are $(\overline{\mathcal{J}} \cap \overline{\mathcal{J}'}) \cup (\mathcal{J} \cap \mathcal{J}')$.

Algorithm 2 detects positive bits (POS), negative bits (NEG) and fixed bits (FIX). We define the status vector $S = \langle S_0, S_1, ..., S_{|B|-1} \rangle$ where each S_j can take one of the three values. The algorithm produces two relations; the first one with all programmable points disabled and the other one with all programmable points activated. If a bit B_i toggles from 0 to 1, it is considered positive. If a bit B_i toggles from 1 to 0, it is considered negative. Bits that don't toggle are considered fixed and remain in the FIX state.

Algorithm 1 can now be modified to take this information into account. A call to `status` must be added at the beginning of this algorithm and the assignment on line 18 must be modified: fixed positions (FIX) are ignored, positive positions (POS) are added to A_i and negative positions (NEG) are added to the inverse address ($A_{\overline{i}}$).

Algorithm 2: Positive an negative positions

```
1  proc status (P, X, Y, S) is
2  begin
3      for j = 0 to |P| − 1 do
4          PX[i] ← X
5          PY[i] ← Y
6      end
7      BX ← BitStreamGenerator(PX)
8      BY ← BitStreamGenerator(PY)
9      for j = 0 to |B| − 1 do
10         if (BX[j]=BY[j]) then
11             S[j] ← FIX;
12         end
13         if (BX[j]=0 and BY[j]=1) then
14             S[j] ← POS;
15         end
16         if (BX[j]=1 and BY[j]=0) then
17             S[j] ← NEG;
18         end
19     end
20 end
```

C. Handling multi-value domains

Algorithm 1 adds positions to A_i that toggle when P_i toggles from X to Y. Let's suppose a 3-valued domain X, Y and Z with the respective coding 01, 11 and 10. When calling the algorithm with values X and Y, only the first address bit is found because only the first bit differs between the codings of X and Y. But, when calling it with values X and Z, all positions are found.

To find all the positions of a given domain, we must call the mapping algorithm with enough cases to ensure that all the related B_j differ in at least one case. But, since we don't know the coding, we cannot determine the minimal set of pairs. As an example, the coding 100, 010 and 001 only needs two calls to find all 3 positions.

A foolproof technique is to try all the possible pairs ($\forall v_1 \in D_i, \forall v_2 \in D_i, v_1 \neq v_2, \langle v_1, v_2 \rangle$). But since all the $C_{i,v}$ for a given i must differ in at least one bit from each other, it is also correct to keep pairs with a common reference (typically, the disabled value if it exists).

Algorithm 3: Multi-value mapping

```
1  proc multi-mapping(P, D, A) is
2  begin
3      for v = 1 to max(|D0|, |D1|, ..., |D|D|−1|) do
4          mapping(P, 0, v, A)
5      end
6  end
```

Algorithm 3 performs multiple calls to the simple mapping procedure. All the programmable points are toggled from the first value to another value in their domain. During the calls to the procedure, the positions that differ in the encoding are accumulated in the address sets. In the end, the address sets contain all possible positions. Algorithm 2 must be extended in the same way.

D. Handling constraints

One must generate consistent configurations for the FPGA to be processed by the *BitStreamGenerator* tool. These configurations must satisfy a set of constraints. Satisfying these constraints is a considerable challenge.

On FPGAs, different kinds of constraints can be found. There are *dependence* constraints which are when a programmable point can only be activated if one of its predecessors is activated. There are *configuration* constraints which are when programmable points depend on some configuration (voltage, I/O protocol, type, ...). There are *conflict* constraints which are when programmable points cannot be activated simultaneously because they share some resources. And finally, there are *share* constraints which are when two programmable points must have a related value (usually a common one) because they share some bitstream bits. We can see all these constraints in Figure 3.

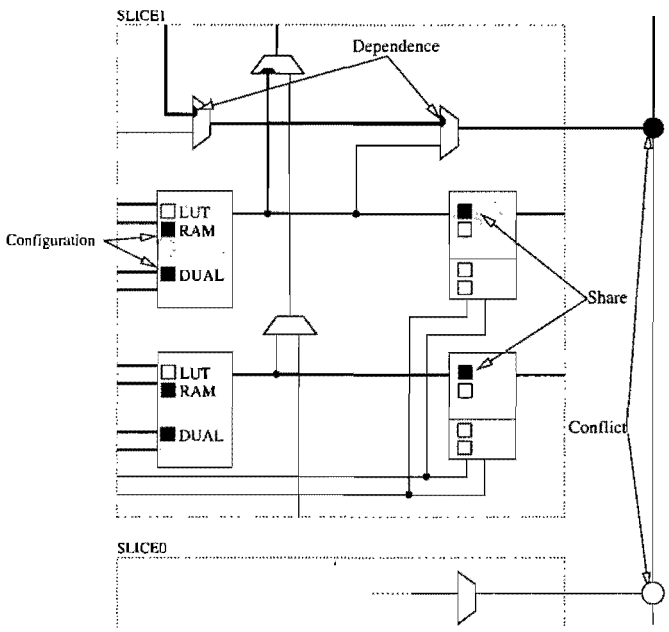


Fig. 3. Illustration of the 4 kinds of constraints

Fortunately, in practice these constraints are local constraints that concern small sets of programmable points whose values are related to each other. The settings of these programmable points can be expressed as the setting of a single new programmable point with an appropriate composed domain. For example, if P_1 and P_2 can each take two values, OFF and ON, but only one of them can have the ON value, we will replace those programmable points with a single one with the composed domain {OFF/OFF, ON/OFF, OFF/ON}. Once all dependencies are removed in this way, our algorith can be applied.

Nevertheless, in our context, it was possible to use a simpler methodology that consists in restricting the domains. D^* is a restricted domain of D when $\forall i\ D_i^* \subseteq D_i$. These restricted

domains limit our algorithm to consistent configurations. By using multiple constrained domains, we can resolve all the programmable points as if we were using unconstrained domains. But we must ensure that we cover all the possibilities for all domains.

Since domain vectors are big, we use a technique to simplify the definition of constrained domains. We use the concept of *preset tiles* to represent partial restrictions. A preset tile contains free and fixed programmable points. We denote a tile by a pair of sets $\langle free, fixed \rangle$. Free programmable points are resolved by our algorithm while fixed programmable points are set to a specific fixed value to have consistent configurations. Other points are unused (unconstrained) by the tile. Generating a constrained domain from a preset tile is accomplished by restricting the domain of the fixed programmable points. For a given set of tiles, programmable points are resolvable when they are already present in one free set.

Supposing we have 6 programmable points $\langle A, B, C, D, E, F \rangle$ with domains $\{\text{OFF}, \text{ON}\}$ and the following constraints:

- A and B cannot be active simultaneously,
- C must be active when A or B is active and
- D and E cannot be active simultaneously.

To solve the mapping, we can use these tiles:

$$\begin{aligned} T_0&: \langle \{A\}, \{B \mapsto \text{OFF}, C \mapsto \text{ON}\} \rangle \\ T_1&: \langle \{B\}, \{A \mapsto \text{OFF}, C \mapsto \text{ON}\} \rangle \\ T_2&: \langle \{D\}, \{E \mapsto \text{OFF}\} \rangle \\ T_3&: \langle \{E\}, \{D \mapsto \text{OFF}\} \rangle \\ T_4&: \langle \{C, F\}, \{\} \rangle \end{aligned}$$

The constrained domains for each preset tile can be generated and solved by using Algorithm 3. The merging of non-interfering tiles can optimize the process. We can merge two tiles if they do not have programmable points in common in any set. As an example, it is possible to merge T_0 with T_2 and T_1 with T_3 and obtain only three tiles.

$$\begin{aligned} T_{0,2}&: \langle \{A, D\}, \{B \mapsto \text{OFF}, C \mapsto \text{ON}, E \mapsto \text{OFF}\} \rangle \\ T_{1,3}&: \langle \{B, E\}, \{A \mapsto \text{OFF}, C \mapsto \text{ON}, D \mapsto \text{OFF}\} \rangle \\ T_4&: \langle \{C, F\}, \{\} \rangle \end{aligned}$$

The merging of tiles and the generation of constrained domains is automatic. The goal of merging is to minimize the number of calls to the *BitStreamGenerator*. The merging problem can be resolved using a graph coloring algorithm (resource allocation). As we are not interested in the optimal merging, we use a simpler linear time technique.

Conflict constraints are most of the time due to multiple drivers on the same wire. This kind of constraint can be handled automatically by an analysis. For each multiple driven wire, we produce a set of tiles (one per programmable point). Each tile contains the programmable point to activate its driver in its free set and all others related programmable points are mapped to the unactivated value in the fixed set.

Dependence constraints can be found automatically by an analysis.

Configuration constraints are handled the same way by using tiles. But, as it is not possible to determine automatically relations between configurations, the process cannot be automatic. We must guide the analysis by manually adding preset tiles which translate the constraints described in the FPGA's documentation. Shared constraints are handled the same way as configuration constraints. But, most of the time, they are not documented. Usually, compilation errors are raised when trying to generate an inconsistent configuration. By looking at error messages, we can determine missing tiles.

Algorithm 4: Preset-tile mapping

```
1  proc tile-mapping(P, D, T, A) is
2  begin
3      while T ≠ {} do
4          D* ← ⟨{}, {}, {}, ..., {}⟩
5          C* ← {}
6          forall ⟨free, fixed⟩ in T do
7              C ← free ∪ {i | (i ↦ j) ∈ fixed}
8              if C* ∩ C = {} then
9                  T ← T / {⟨free, fixed⟩}
10                 C* ← C* ∪ C
11                 forall (i ∈ free) do
12                     D*_i ← D_i
13                 end
14                 forall (i ↦ j) ∈ fixed do
15                     D*_i ← {j}
16                 end
17             end
18         end
19         multi-mapping(P, D*, A)
20     end
21 end
```

V. A CASE STUDY: XILINX VIRTEX-II PRO

Xilinx is a leader in the manufacturing of reconfigurable device. The Virtex-II Pro has an embedded PowerPC processor and is equipped with the Internal Configuration Access Port (ICAP), which enables dynamic self-reconfiguration. These features makes this technology very attractive to study innovative techniques and tools related to dynamic designs. Virtex-4 and Virtex-5 devices now also offer such capability. The proposed methodology could easily be adapted to these devices. As mentioned in the introduction, we have focused on finding the mapping of CLBs, BRAMs and multipliers because the other resources are not useful in the context of JIT HC. In this section, we present the information required to target Xilinx devices and how it can be found. We also describe how we adapted our algorithm to Xilinx devices and tools.

A. Requirements

Our methodology requires the following information and tools:

1) A detailed description of the FPGA's programmable points.
2) A way to generate the bitstream from a set of programmable points.

3) The possibility to extract the vector of bits out of the bitstream.

In the following subsections, we present how these requirements can be satisfied in the context of Virtex-II Pro devices.

1) Detailed description of the programmable points: Virtex-II Pro devices are documented in [Xil07a], [Xil07b], which contain a high level description of the FPGA components. This documentation is not sufficiently detailed but it is helpful to understand some of the constraints. A more detailed description can be obtained by using the Xilinx provided XDL tool when used to produce a report of a given device. Figure 4 contains a simplified example of an XDL report.

```
1. (xdl_resource_report v0.1 xc2vp2fg456-6 virtex2p
2.  (tiles 23 35
3.  [...]
4.  (tile 2 2 R1C1 CENTER 8
5.    (primitive_site VCC_X1Y16 VCC internal 1 -1
6.      (pinwire VCCOUT output VCC_PINWIRE))
7.    (primitive_site SLICE_X0Y30 SLICE internal 47 4
8.      (pinwire BX input BX_PINWIRE0)
9.      (pinwire BY input BY_PINWIRE0)
10.     (pinwire CE input CE_B0)
11.     (pinwire CIN input CIN0)
12.     (pinwire CLK input CLK0)
13. [...]
14.   (wire ALTDIG0 0)
15.   (wire ALTDIG1 0)
16.   (wire ALTDIG2 0)
17.   (wire ALTDIG3 0)
18.   (wire E2BEG0 2
19.     (conn R1C2 E2MID0)
20.     (conn BRAMR2C1 E2END_S0))
21. [...]
22.   (pip R1C1 W6END_N8 -> W6BEG0)
23.   (pip R1C1 W6END_N8 -> S6BEG0)
24.   (pip R1C1 W6END_N8 -> N6BEG0)
25.   (pip R1C1 W6END_N9 -> W6BEG1)
26.   (pip R1C1 W6END_N9 -> S6BEG1)
27.   (pip R1C1 W6END_N9 -> S2BEG0)
28. [...] (primitives_def
29.  [...]
30.  (primitive_def SLICE 47 136
31.    (pin BX BX input)
32.    (pin BY BY input)
33.    (pin CE CE input)
34.    (pin CIN CIN input)
35.    (pin CLK CLK input)
36. [...]
37.    (element FXMUX 4
38.      (pin F5 input)
39.      (pin F input)
40.      (pin FXOR input)
41.      (pin OUT output)
42.      (cfg F5 F FXOR)
43.      (conn FXMUX OUT ==> XUSED 0)
44.      (conn FXMUX F5 <== F5MUX OUT)
45.      (conn FXMUX F <== F D)
46.      (conn FXMUX FXOR <== XORF 0))
47. [...] (summary tiles=805 sites=3566 sitedefs=27
48.          numpins=83339 numpips=1709085))
```

Fig. 4. An XDL report for Virtex-II Pro (VP2)

The report contains device information (line 2), tiles (line 3) and primitive definitions (line 29). A device is a grid of tiles containing interconnected primitives (e.g. SLICE, TBUF, ...). In the example, the FPGA is made of a 23x35-tile grid. The relation between the FPGA and XDL is shown in Figure 5

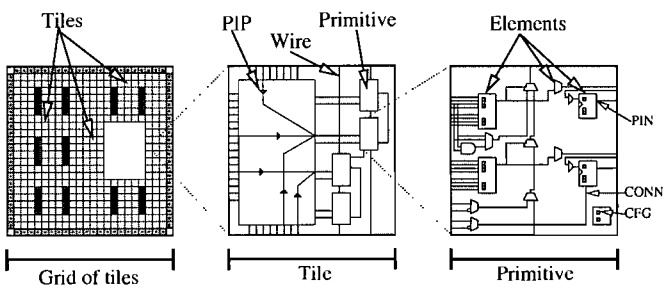


Fig. 5. Relation between FPGA and XDL

Each tile is declared after the grid dimensions. Line 4 contains the declaration of a CENTER tile (CLB) named R1C1 which is located at position (2,2) in the grid. Tiles contain primitive instantiations. For example a CLB on Virtex-II Pro contains 4 slices (line 7 instantiates one of these). T instantiation declares the pinout and connections to local wires. A tile also contains wires that are connected by connections (CONN) and Programmable Interconnect Points (PIP), which are configurable connections between wires.

Primitive definitions follow the tiles declaration (line 29). They are used as templates for primitive instantiations. Definitions contain the pinout and elements. Elements are the basic blocks (multiplexers, registers, lookup tables, ...) and their behaviors are not defined in the XDL report. At line 37, there is a programmable element FXMUX which can be configured with values F5, F or FXOR (line 42).

There are two kinds of programmable points: PIP and configurations (cfg). For PIP, the domains contain two values: active and inactive. Domains for configurations are more complex. They are defined in the report.

We have shown that it is possible to enumerate the programmable points and their domains (D) by using an XDL report in a relatively simple way, which satisfies our first requirement.

2) Bitstream generation: Design implementation is the process that transforms a circuit description into a bitstream for a given circuit. This process can be divided in different stages that are implemented by specific tools. Figure 6 shows the standard design flow when using the Xilinx ISE Tools.

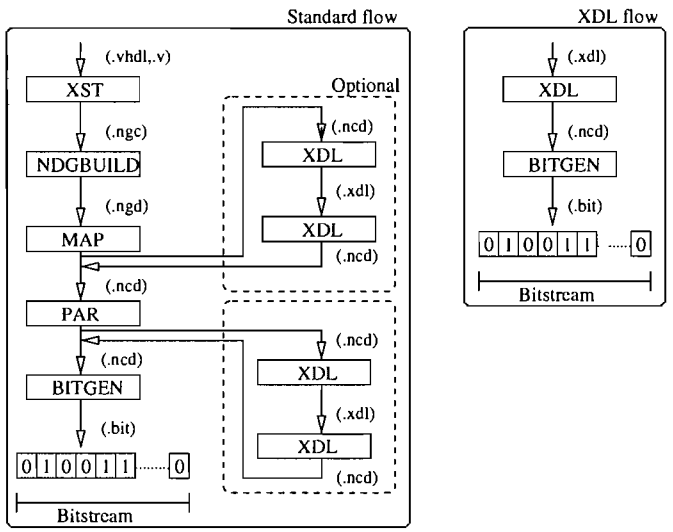


Fig. 6. The standard design flow and the XDL flow

These tools can be used from the ISE graphical user interface, from the command lines or by an external script. They perform HDL parsing (xst), RTL synthesis (ngdbuild), technology mapping (map), place and route (par), and finally the bitstream generation (bitgen). For our purpose, we need a fine control on the configuration points and these stages don't allow it. Nevertheless, Xilinx also provides a way insert third party tools in the design flow through a proprietary language: XDL (Xilinx Description Language) as illustrated in the dotted boxes of Figure 6. An example of a circuit representation in this language is shown in Figure 7.

```
1. design "dummy" xc2vp2fg256-6 v2.38 ;
2. inst "FUNC" "SLICE",
3.  placed R16C21 SLICE_X41Y1, cfg "YUSED::0
4. XUSED::0 F::#LUT:D=A1+A2
5.      FXMUX::F SYNC_ATTR::ASYNC GYMUX::G
6.      G::#LUT:D=A1*A2 _SUPERBEL::TRUE";
7. [...] net "net1",
8.  cfg "_NET_PROP::IS_BUS_MACRO:",
9.  inpin  "FUNC" F1,
10.  inpin  "FUNC" G1, outpin  "FUNC" Y,
11.  pip LIOITTERM TTERM_N2MID3 ->
12. TTERM_S2END8, pip LIOITTERM TTERM_N2BEG7
13. -> TTERM_S2MID2, [...] #
14. ;
```

Fig. 7. XDL file produced by ncd2xdl

An XDL circuit description is composed of a header (line 1), instance declarations (line 3) and net declarations (line 8). An instance of a primitive is declared by the *inst* syntax and is named by the user in the circuit description. It may be placed manually by specifying the primitive location described in the XDL report (R16C21 SLICE_X41Y1). Elements are configured in the *cfg* string (line 4). In the example, element FXMUX is configured to the value F and nets are declared with the *net* syntax (line 8). It contains pins (inputs and outputs) and PIPs activated to route the signal.

To satisfy our second requirement, we propose to generate straightforwardly an XDL representation of the programmable points' settings and use the XDL flow, which is described on the right of Figure 6, to produce the bitstream.

3) Bit vector extraction: Virtex-II Pro configuration relies on a packet processor. The packet processor is a state machine with a set of registers that drives incoming data into the target configuration register. Bitstream data packets consist of a 32 bit header and a body of variable length. There are two kinds of packets: type 1 and type 2.

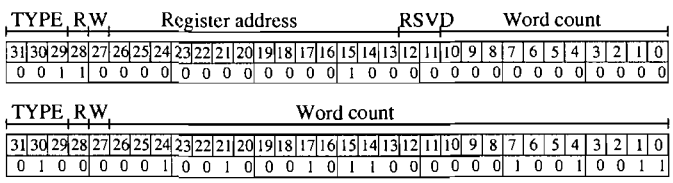


Fig. 8. Bitstream packet types 1 and 2

Type 1 packets can contain a maximum of 2048 data words. Type 2 packets are preceded by a type 1 packet and can contain much more data. The R and W bits indicate the action of the packet (read or write) on the register specified in the register address. Data words follow the packet header. The word count indicates the number of words to process.

```
1. (DUMMY)
2. (SYNC)
3. (WRITE CMD #u32(7) RCRC)
4. (WRITE FLR #u32(45) ())
5. (WRITE COR #u32(278501))
6. (WRITE IDCODE #u32(19030163) ((device 2vp2)))
7. (WRITE MASK #u32(0) ())
8. (WRITE CMD #u32(9) SWITCH)
9. (WRITE FAR #u32(0) ((BA 0) (MJA 0) (MNA 0)))
10. (WRITE CMD #u32(1) WCFG)
11. (WRITE FDRI #u32(0 0 0 0 0 0 ....))
12. (auto-crc #u32(10304))
13. (WRITE CMD #u32(10) GRESTORE)
14. (WRITE CMD #u32(3) LFRM)
15. (NOOP)
16. [...]
17. (NOOP)
18. (WRITE CMD #u32(5) START)
19. (WRITE CTL #u32(0)) (WRITE CRC #u32(24407))
20. (WRITE CMD #u32(13) DESYNCH)
21. (NOOP)
22. [...]
```

Fig. 9. Decoded bitstream

Figure 9 is a decoded representation of a bitstream generated by our tool. Each line is a packet with an action (read/write) and the corresponding register. The data words follow in a vector of unsigned 32 bit integers. The rest of the line is a decoded version of the data words.

Bits which are used to configure the FPGA are written to the FDRI (Frame Data Register Input) register at addresses specified by the FAR (Frame Address Register) register. Without the compression option, bitgen generates only one packet that writes to the FDRI, which contains all the configuration bits. It is easy to extract the bits from this packet (see line 11) and build our bitstream vector B.

This last point demonstrates that we are capable of fulfilling the three requirements in the context of Virtex-II Pro devices.

B. Application of the proposed methodology

Our methodology applied to the Virtex-II Pro technology consists in generating XDL files implementing our programmable point vector P, processing them with bitgen and finally extracting bits from the resulting bitstreams to build our bitstream vector B. As detailed in Section IV the mapping between P and B can be resolved as soon as each P_i can be isolated. This is challenging because we can only produce valid configurations. This is why we have introduced the concept of tile in the same section.

In the context of Xilinx devices, we used different approaches to produce the tiles for network (PIPs) and configurations. We want an automatic way to produce these tiles and when this is not possible, we want a general and simple way to minimize human effort (and errors).

1) Network: The first kind of tiles used by our algorithm is for networking. We have faced two problems related to nets: simplification and multiple drivers.

Bitgen does not directly produce a bitstream from a configuration vector because it performs some sanity checks and simplifications. Unconnected nets and useless configurations are simplified. A way to avoid simplification is to produce connected nets. We think this task could be automated but we used a simpler approach. It is possible to use a special annotation (line 9 of Figure 7) to specify to bitgen that a net is used by dynamic reconfiguration. This way unconnected nets are not simplified.

The other problem occurs when there are multiple drivers on the same segment (connected wires and connections). This is equivalent to a short circuit and can damage the chip. Bitgen's behavior is not the same on ISE6, ISE7 and ISE8. Older version crash while newer versions report the problem or simplify the circuit (by disconnecting all the drivers). To automatically produce valid configurations, we perform a labeling phase and a graph coloring of the wires. The labeling phase consists in finding a unique name for all connected wires and connections (segments). This way by looking at wire names of PIPs we know if they drive the same segment. The second phase consists in splitting the tile into sub-tiles free of conflicts. This is achieved by a heuristic graph coloring that distributes labeled segments in sets where each segment cannot have more than one driver for the same label. Each set can then trivially produce a tile and be used by our algorithm.

The production of configurations for networking is completely automatic and does not need human intervention. This technique seems to be possible for all the Xilinx devices but we did not perform exhaustive tests.

For each PIP we get a set of addresses modified when toggling it. By looking at our results, we found overlapping address vectors for different PIPs. We observed that these overlapping PIPs always drive the same wire and we deduced that these PIPs are dependent. Theoretically, they must be considered as a programmable vector with multiple values. We used a simpler approach that consists in merging the addresses found for each PIP of a programmable vector since we already calculated them using Algorithm 1. Since we know that only one PIP in the vector can be activated at a time, we can deduce the encoding of each PIP with the merged addresses set.

PIP			A_i	Coding
R1C1 X3	→	E2BEG0	{64951, 67892}	0100 1000
R1C1 Y0	→	E2BEG0	{64951, 67895}	0100 0100
R1C1 N2MID0	→	E2BEG0	{66421, 69364}	0001 0010
R1C1 S2END2	→	E2BEG0	{64951, 69364}	0100 0010
R1C1 S2MID0	→	E2BEG0	{66420, 69364}	0010 0010
R1C1 N6END0	→	E2BEG0	{66421, 67892}	0001 1000
R1C1 S6MID0	→	E2BEG0	{64951, 69367}	0100 0001
R1C1 N2END_N9	→	E2BEG0	{66421, 69367}	0001 0001
R1C1 N6MID0	→	E2BEG0	{66420, 69367}	0010 0001
R1C1 OMUX_E2	→	E2BEG0	{64950, 67895}	1000 0100
R1C1 OMUX_EN8	→	E2BEG0	{64950, 67892}	1000 1000
R1C1 S6END1	→	E2BEG0	{66420, 67895}	0010 0100
R1C1 E6END0	→	E2BEG0	{66420, 67892}	0010 1000
R1C1 E2END0	→	E2BEG0	{64950, 69364}	1000 0010
R1C1 E2END2	→	E2BEG0	{64950, 69367}	1000 0001
R1C1 W6END0	→	E2BEG0	{66421, 67895}	0001 0100
Merged addresses set:			{64950, 64951, 66420, 66421, 67892, 67895, 69364, 69367}	

Fig. 10. Encoding of the PIPs which drive the same wire (E2BEG0)

Figure 10 shows resulting encoding for PIPs that drive wire `R1C1 E2BEG0` and the corresponding addresses vector. As an example, our algorithm found 2 positions for the PIP `R1C1 X3 → E2BEG0` and the encoding can be deduced from the merged addresses set.

We discovered that some PIPs do not map to any address. As an example, the PIP `R1C1 W6BEG5 → LH12_TESTWIRE` is the only PIP that drives wire `LH12_TESTWIRE` and it does not toggle any bit in the bitstream. These PIPs are always activated and are for internal use only (perhaps as sanity checks).

2) Configurations: Configurations of elements are also limited by constraints. Inconsistent configurations that do not pass the design rule check (DRC) are not generated by bitgen. Sometimes some configurations are simplified without warning. We did not find any annotation to avoid simplification of configurations. In this case, the only way we found was to write tiles manually.

The required number of tiles is reasonably small because all primitives are identical. By using this documented regularity, we can generate tiles from templates. Figure 11 shows two templates used for tiles having type SLICE. The first one resolves `CYINIT` and the second one resolves `CY0F` and `CY0G`. We wrote 11 templates by hand to resolve SLICE. This is probably not the minimal set but works fine.

Free Elements	Fixed elements
CYINIT	BXINV::BX BXOUTUSED::0 CYSELF::1 CYSELG::1 COUTUSED::0
CY0F CY0G	CYINIT::CIN COUTUSED::0 BXINV::BX BYINV::BY BXOUTUSED::0 BYOUTUSED::0 CYSELF::F CYSELG::G FXUSED::0 FXMUX::F GYMUX::G XUSED::0 YUSED::0 F::#LUT:D=0 G::#LUT:D=0

Fig. 11. Slice configuration presets

Configuration points have domains with multiple values. When performing the analysis we must keep track of the encoding.

Element	A_i	Value		Coding
DXMUX	{44284}	1	↦	1
		0	↦	0
CEINV	{50174}	CE_B	↦	0
		CE	↦	1
CLKINV	{50156}	CLK_B	↦	1
		CLK	↦	0
FXMUX	{48678, 48734}	F	↦	00
		FXOR	↦	10
		F5	↦	01

Fig. 12. Encoding of some programmable points of R1C1 SLICE_X1Y31

As we can see in Figure 12, encodings are not the same for different (and similar) elements.

VI. Results

We have applied our methodology to a Xilinx Virtex-II Pro device (XC2VP2). Our implementation is highly parallel. Solving the mapping for the XV2CP2 took four computer-days and required 2282 invocations of bitgen. We fully determined the mapping of the networks (PIPs) and configuration points of components (slices, tbuf, ...) for all CLBs, BRAMs and multipliers. We also successfully solved the mapping of a Spartan3 device in a day to demonstrate that the technique is also applicable to other FPGAs.

Each position in the bitstream has an absolute and a relative address. A relative address is composed of a block address, a major frame address, a minor frame address and the offset. The Virtex-II Pro has three block addresses: CLB, BRAM-Interconnect and BRAM.

Our algorithm finds the absolute addresses. By calculating the corresponding relative addresses, we can represent our results in a graphical way. This is illustrated in Figure 13, which shows the address area whose mapping has been found. The black pixels represent bits of unknown mapping while the gray and white pixels are known.

A column of pixels is a minor frame where each pixel maps a bit in the bitstream. The minor frames are grouped in wider columns to form major frames, which are clearly visible in the CLB address space. The first major frame configures global components such as the clocks. The second and the last major frames configure the IOB. In the middle of those columns, we can observe the 22 major frames, each consisting of 22 minor frames, which correspond to the FPGA CLBs. We

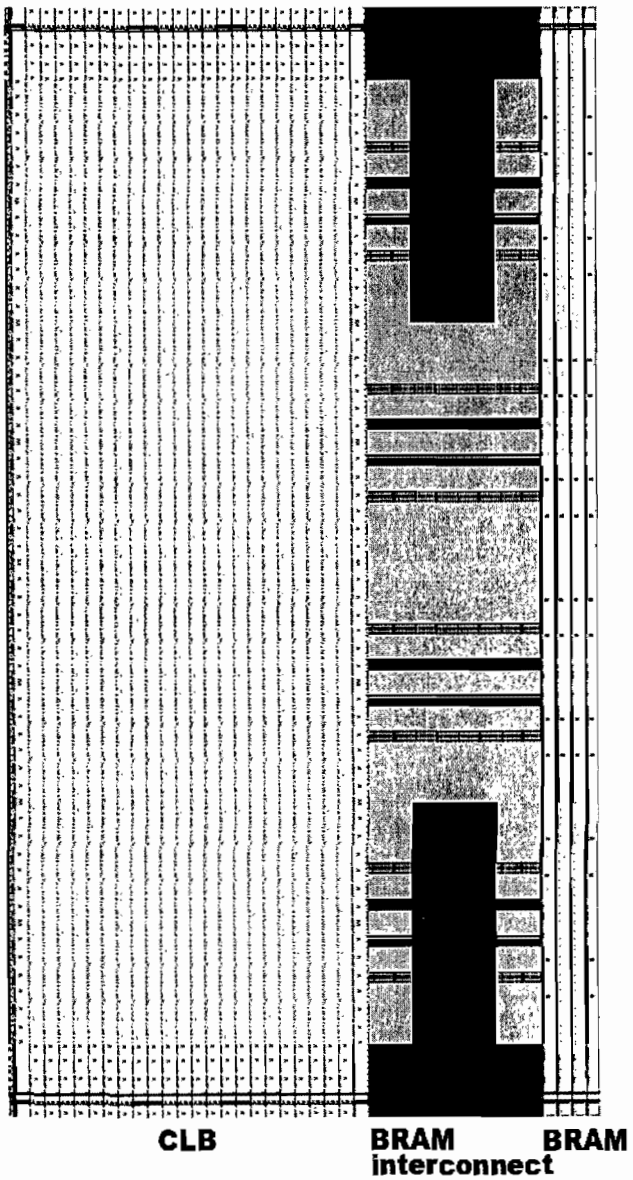


Fig. 13. Graphical map of XC2VP2 bitstream frames

can differentiate the network from the configuration bits by the white and gray colors respectively. The configuration bits are the first minor frames of the CLB columns.

In Figure 13, we can see that the first major frame is incomplete, as well as the BRAM and BRAM-Interconnect blocks. This phenomenon can be explained by the fact that we have not implemented the required tiles for global components and specialized sites. Essentially, we focused on the sections required by our final goal: JIT HC.

It is important to note that our algorithm does not assume any regularity in the structure of the FPGA. Other approaches, which rely on the relatively small number of configuration bits for a given sub-block, would indeed fail to find the mapping of a large block since the complexity is exponential in the number of configuration bits. Our algorithm, due to its logarithmic complexity, can find the mapping of large and non redundant blocks. Evidently, our results do confirm the regular structure of the Virtex-II Pro. The results obtained by the calculation of the relative addresses match the Xilinx documentation on the configuration of the Virtex-II Pro [Xil07b] and confirm the applicability of our technique. An interesting point is that we have detected a few exceptions in this regularity, which would not have been possible if we had assumed it.

VII. Tools and Applications

Our initial goal was to dynamically synthesize, place and route logic and arithmetic expressions to implement JIT HC. This point is detailed in Section VII-E. Nevertheless, our methodology has enabled many other research avenues and the development of novel tools. This section briefly describes these applications, which are currently in the state of work in progress.

A. Regenerating XDL from bitstream

Once we are able to extract the configuration of the programmable points from a bitstream, we can try to generate an XDL file from the obtained configuration. An FPGA is almost never used at it's full capacity and many resources are actually unused. An XDL file generated blindly from this configuration data would fail the DRC. The typical errors are components with unconnected inputs or outputs, useless PIPs in a net, etc.

However the Xilinx tools will accept to convert an XDL file that contains design errors by using a special command line switch. But even if a blindly generated file can be converted to a ".ncd" file, it would not be manageable because all the sites of the FPGA would be instantiated while most of them would be unused (just containing a default configuration). Therefore, we need to find a way to purge the FPGA configuration file from all useless data.

We have written a useful element collector (UEC) algorithm to fix this problem. This algorithm propagates the usefulness property from the outputs to the inputs of all the elements and PIPs. When the UEC runs through an element or a wire, it is marked as *useful*. The list of the useful wires is used to generate the list of the nets included in the configuration by means of the union-find algorithm. The used sites can be deduced from these nets since a site needs to be connected to a net to be used. Finally, we remove all the configuration data that is useless and we produce a valid XDL file, free of DRC errors and with exactly the same semantics as the original design.

To validate our results, we generated a simple circuit (32 bit clocked adder) using the normal Xilinx design flow. The generated circuit can be viewed in Figure 14. With our tools, we generated a new XDL (".xdl") file from the corresponding bitstream (".bit"). We observed that the original and the regenerated circuits were identical, which shows that it is possible and quite easy to reverse-engineer circuits whose bitstream is not encrypted.

To confirm our approach we designed two simple circuits in VHDL, a clocked multiplier (8x8 $\rightarrow$ 16 bits) and a 16 bit CRC generator. And we generated their bitstreams using the normal Xilinx design flow. After this, the bitstreams were processed with our tool to obtain XDL descriptions.

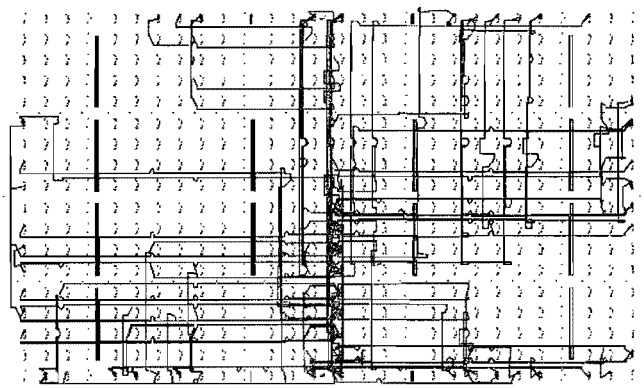


Fig. 14. FPGA editor screenshot of a reversed bitstream of a 32 bit adder

Finally, new bitstreams have been regenerated from these XDL descriptions using standard Xilinx tools. We noticed that the original bitstreams were identical to the regenerated ones.

B. Low level FPGA simulator

We created an FPGA simulator that is capable of simulating a Virtex-II Pro device from its configuration bitstream. This event driven simulator is implemented using simple FPGA elements such as multiplexers, inverters, XOR/AND/OR gates, LUTs, etc. All the components are connected together as described in the XDL device report.

The Xilinx tools can simulate static designs but cannot cope with dynamic behavior. This limitation makes it hard to debug these kinds of applications. The end objective of this project is to demonstrate the simulation of dynamically reconfigurable designs.

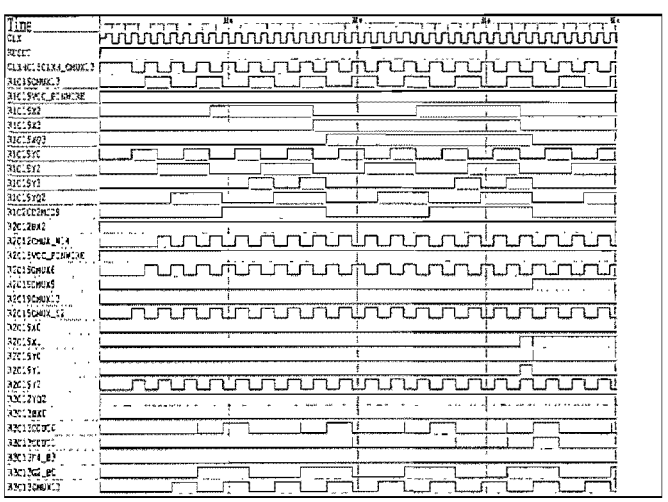


Fig. 15. Low level simulation of a 32-bit adder

The simulator produces Value Change Dump (VCD) files, which are readable by standard tools. In our experiments, we were able to easily simulate an FPGA implementation of an 8-bit counter and a CRC generator/validator circuits. We are now planning the simulation of an FPGA implementation of a microcontroller. As mentioned earlier, this simulator is still a work in progress, especially for the implementation of the FPGA sites. With some modifications, our simulator will fu simulate the FPGA with dynamically modified bitstreams.

C. Custom Bus Macro

Bus macros [LP02] are used as a communication channel between dynamic regions. On the Virtex-II Pro, they are implemented with tri-state buffers. As there are only two TBUF in a CLB, the limitation of 2 bits per CLB is often a bottleneck in the communication between modules. However, the use of tri-state buffers is not a physical limitation but a software one. The interface routing must be the same for all the instances of a dynamic module. With the Xilinx tools, it is possible to specify constraints on the placement of components but not on the routing. A bus macro forces the placement of two tri-state buffers. As there is only one path possible between these components, all instances have the same routing. This way, interfaces are compatible when a dynamic module change occurs.

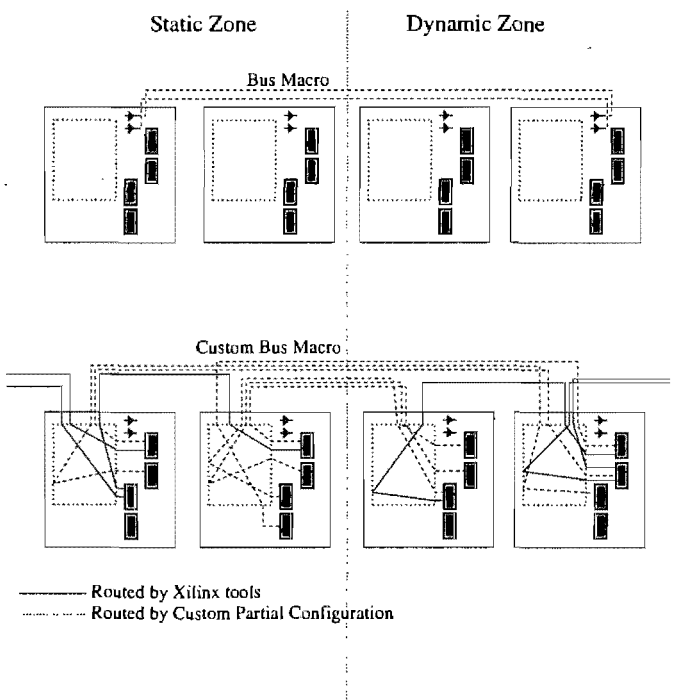


Fig. 16. Custom bus macro interfacing

To overcome this limitation and still use the Xilinx tools, we produce custom bus macros (Figure 16). The general idea is to produce the modules without connecting them. Then a *link* phase connects the nets between the modules just after their instantiation. A similar idea was implemented in [RW07]. But instead of parsing directly the bitstream, they convert the design using the xdl tools and parse the textual representation of the design.

Some constraints force the placement of the communicatio.. logic between the modules to be near the boundaries. By using a custom tool, we produce a partial configuration for each instance of the dynamic module. This partial configuration

must then be merged with the partial configuration of the static module.

In our prototype, we force the communication to be latched and place some registers on the boundaries between the static and the dynamic parts (placement constraints). We parse the static bitstream and the dynamic bitstreams of the modules to determine which wires are used. Finally, we use a routing algorithm to find a path between the registers and we produce the partial bitstream to configure these paths.

The technique of custom bus macros [RW07] requires static analysis of the XDL representations of modules. An advantage of producing bus macros directly from the bitstream is that we can also produce them for dynamically generated modules.

D. Abstract Annotated Tiles

Typically, the granularity of the components handled by Run-Time Reconfigured (RTR) systems is the module. This granularity is not fine enough to realize a JIT that needs basic instructions (such as arithmetic operators, logic operators, multiplexers, ...). To fix this problem, we proposed *annotated tiles* [BFD07]. The idea is to provide a set of fine-grained tiles annotated with the information necessary to handle them correctly.

Figure 17 shows a pipeline built by merging basic tiles. To be able to produce this kind of module, tiles must be able to overlap and cannot pass through bus macros.

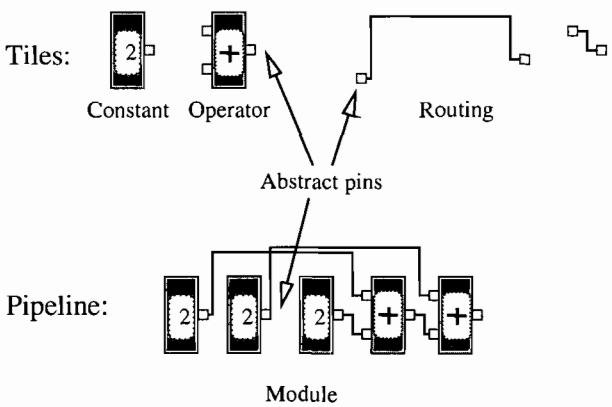


Fig. 17. Construction of a pipeline based on annotated tiles expression 2+2+2

With the information obtained by the proposed technique, we produce tiles without using the Xilinx tools. Instead of representing a tile as a rectangular set of bits, our tiles contain a mask to specify which bits are actually used. This mask can be used to merge overlapping tiles. Tiles can be merged if and only if they do not use common resources. *Abstract pins* are used to represent a set of interconnection points. Tiles produced by our tool are annotated with such pins to specify their interfaces.

These tiles can be used as basic blocks by a hardware compiler. The compiler has to connect basic tiles together without conflict to implement the required expression. Figure 18 shows an example of dynamically generated cores by using a set of abstract annotated tiles.

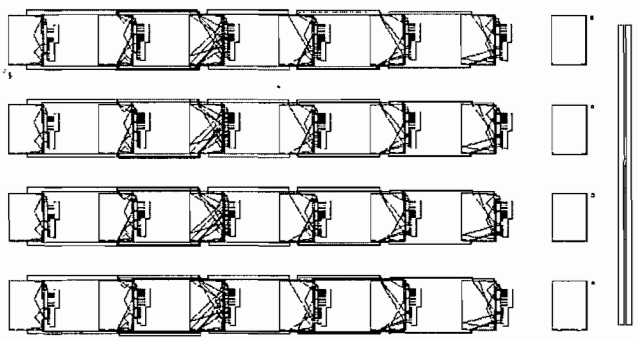


Fig. 18. Dynamic design produced with abstract annotated tiles

E. JIT Hardware Compilation

JIT compilation can be seen as code dynamically translated and optimized for the target architecture. The JIT HC [BFD08] shares the same idea but instead of producing code, the compiler produces a partial configuration for an FPGA. Because specialized modules are faster, some applications should have speedups when using this kind of execution by increasing the functional density [WH97].

There are two major problems with JIT HC. The first one is algorithmic. In the general case, JIT HC needs a fast place and route algorithm. The VPR [BR97] and River Side On-Chip Router (ROCR) [LVT04] are some examples of research projects trying to address these problems efficiently. The second problem is the lack of information regarding the target circuits. Until now, it has not been possible to write a JIT backend without using the Xilinx tools. In order to investigate JIT HC, we implemented a prototype system on the Xilinx ML310 demo board containing a VirtexII-Pro VP30.

Figure 19 is an FPGA editor screenshot of the design we implemented to support JIT HC. The left part of the design is the static part of the system. The right part has two dynamic zones (upper and lower) that will be filled with the modules produced on-the-fly by our compiler running on the PowerPC. Each zone has its own interface *Bus Macro* to the static zone.

We ported the Gambit-C interpreter for the Scheme language [KCE98] to run on the embedded PowerPC. We added the `synthesize` primitive, which translates a functional closure to a hardware pipeline. Finally, we used the bitstream information to write a JIT backend. Our compiler implements several phases to parse a Scheme expression and produce the partial bitstream. The design is able to load them dynamically through the *ICAP* module.

The most intricate part is the place and route algorithm. Currently we are able to synthesize simple designs in less than 20 ms (placement: 5 ms, routing 15 ms). More complex designs such as the 28-stage MD5 hash calculator depicted in Figure 20 may require up to 500 ms. We limited this design to 28 stages (instead of 64) because the number slices available in the dynamic zone did not permit a full implementation. These results are very preliminary and must certainly not be considered as the best performances achievable. They are reported here just to convince the reader that the concept of

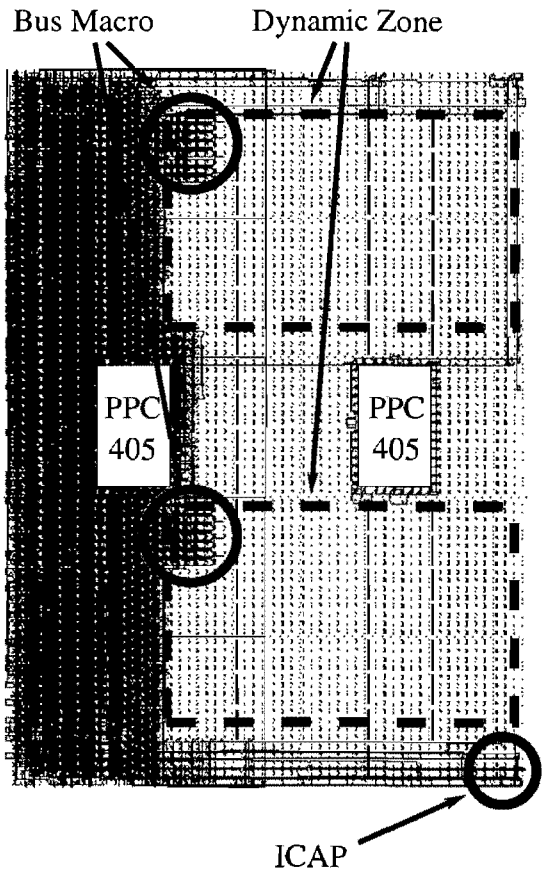


Fig. 19. Design to support JIT (Virtex-II Pro VP30)

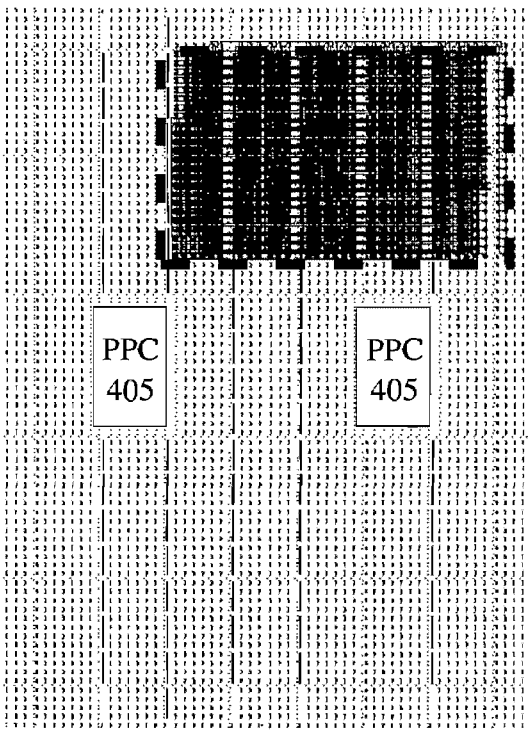


Fig. 20. 28-stage MD5 dynamically produced by a JIT

JIT HC is promising and that this work opens new avenues in the field of hardware accelerators.

VIII. Conclusion

We have presented a methodology to determine the mapping between the relevant parts of an FPGA bitstream and its programmable points in order to implement JIT hardware compilation. This methodology only requires a detailed description of the FPGA's programmable points, a tool to generate the bitstream from them and a way to access the bits. Thanks to the use of an algorithm with logarithmic time complexity, we have been able to determine the bitstream format of a real device in a few days on a single computer.

This information was necessary to further investigate the field of JIT hardware compilation. Other topics such as dynamically reconfigurable design simulation, dynamic connection of pre-compiled modules etc. are also concerned. In addition to the proposed methodology, our results demonstrate that it is now possible to perform JIT hardware compilation inside an FPGA and to simulate dynamically reconfigurable designs.

An important byproduct of this research is a demonstration that plain bitstreams do not protect the IP of a circuit, even if the bitstream format is not documented by the manufacturer. Given that there are cryptographic ways to protect the bitstream, we urge the manufacturers to fully document the format of their bitstreams to allow third-party design tools and to open new and promising opportunities in the field of dynamically reconfigurable hardware.

References

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent Dynamic Optimisation System. Technical report, Hewlett-Packard Labs, 2000.

[BFD07] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Toward On-Chip JIT Synthesis on Xilinx Virtex-II Pro FPGAs. In *50th International Midwest Symposium on Circuits and Systems/5th International Northeast Workshop on Circuits (MWCAS/NEWCAS)*, August 2007.

[BFD08] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs. In *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2008.

[BH98] Peter Bellows and Brad Hutchings. JHDL - An HDL for Reconfigurable Systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184. IEEE Computer Society Press, 1998.

[BR97] Vaughn Betz and Jonathan Rose. VPR: A new Packing, Placement and Routing Tool for FPGA Research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.

[Fra03] Neil Franklin. VirtexTools FPGA Programming and Debugging Tools Project. `http://neil.franklin.ch/Projects/VirtexTools/`, 2003.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[HL01] Edson Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, July 6, 2001.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[LP02] Davin Lim and Mike Peattie. Two Flows for Partial Reconfigurable Core Based on Small Bit Manipulations, XAPP290 (v1.0). Technical report, Xilinx, May 2002.

[LSV06] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp Processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(3):659–681, 2006.

[LVT04] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 954–959, New York, NY, USA, 2004. ACM.

[NER08] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In *FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 264–264. ACM, 2008.

[Not07] Jean-Baptiste Note. FPGA Netlist recovery. `http://www.ulogic.org`, 2007.

[PHP+04] Alexandra Poetter, Jesse Hunter, Cameron Patterson, Peter Athanas, Brent Nelson, and Neil Steiner. JHDLBits: The Merging of Two Worlds. *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications (FPL 2004)*, Aug 2004.

[PHPA04] Alexandra Poetter, Jesse Hunter, Cameron Patterson, and Peter Athanas. JHDLBits. *Proceedings of the 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2004)*, Jun 2004.

[RS02] Anup Kumar Raghavan and Peter Sutton. JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 192, Washington, DC, USA, 2002. IEEE Computer Society.

[RW07] S.J. Raaijmakers and S. Wong. Run-Time Partial Reconfiguration for Removal, Placement and Routing on the Virtex-II Pro. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007.

[SP02] Reetinder P. S. Sidhu and Viktor K. Prasanna. Efficient metacomputation using self-reconfiguration. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 698–709, London, UK, 2002. Springer-Verlag.

[Ste02] Neil Joseph Steiner. A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, August 2002.

[Sun99] Sun. The Java Hotspot Performance Engine Architecture. Technical report, April 1999.

[WH97] M. J. Wirthlin and B. L. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, 1997.

[Xil07a] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide, UG012 (v4.2). Technical report, Xilinx, November 2007.

[Xil07b] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, DS083 (v4.6). Technical report, Xilinx, March 2007.

Toward On-Chip JIT Synthesis on Xilinx VirtexII-Pro FPGAs

Etienne Bergeron, Marc Feeley, Jean Pierre David
Université de Montréal, École Polytechnique de Montréal

Abstract— **Xilinx VirtexII Pro FPGAs support dynamic reconfiguration. To benefit from this functionality, Xilinx proposes a *modular* and *differential* development flow, which consists in pre-compiling all possible configurations and switching from one to another in real time. The pre-compilation process is too slow and static. Xilinx also supplies *JBits*, but this tool does not support the VirtexII Pro FPGA and later devices. We aim to dynamically produce digital circuits. Unfortunately, since Xilinx does not entirely document the format of the FPGA bitstreams, it is in principle impossible to produce bitstreams without using their tools. This paper presents the methodology we have used to determine the Xilinx bitstream format in order to quickly produce valid configurations on the fly using only our tools. Our synthesis approach translates a simple expression language into a dataflow graph of predefined tiles which are placed and interconnected using the bitstream format information we gathered.**

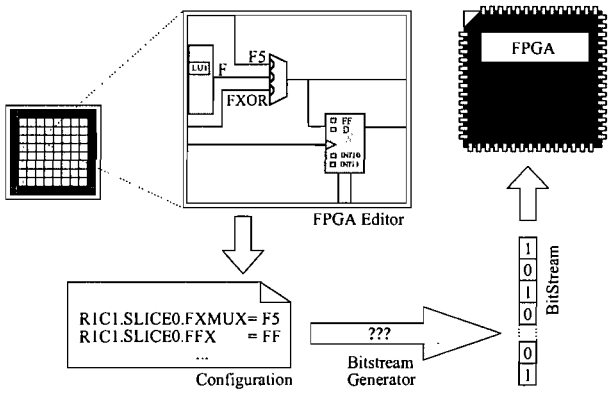


Fig. 1. Mapping Configuration and Position

I. Introduction

A FPGA is a configurable circuit. Its behavior can be tailored to a specific application through the process of *configuration*, which is performed when the FPGA is initially powered on. Some FPGAs, such as the VirtexII Pro family, also allow partial reconfiguration at runtime [Xil05b], [Xil05a].

A configuration is a set of activated programming points. In the case of Xilinx FPGAs, these are either configuration points (LUT, registers, multiplexers) or *programmable interconnect points* (pips). The configuration completely defines the circuit's behavior at the lowest level of abstraction possible for a given FPGA.

A bitstream is a vector of bits encoding a configuration. It is downloaded into the FPGA during the configuration process and it sets up the configuration registers according to the configuration. Xilinx does not document the mapping between a configuration and its associated bitstream.

II. Related work

Bitgen, a Xilinx tool, takes a configuration and produces a bitstream. It is the only tool that can produce bitstreams for the VirtexII Pro FPGA series. JBits [GLS99], another Xilinx tool, can produce bitstreams programmatically from Java, but it does not support VirtexII and later FPGA families. At this time, reconfigurable computing projects on Xilinx FPGAs must use the Xilinx development flow [Xil04] due to the lack of alternatives. The creation of custom tools is not feasible because the relation between the configuration points and the position and encoding of the programming bits in the bitstream is not documented.

PARBIT [HL01] is a tool which extracts and reallocates Virtex partial bitstreams. It can produce a partial bitstream from a specified rectangular area in the CLB region of a full bitstream. It does not support the VirtexII (Pro) family. BITPOS (BITstream POSitioner) [KJdlTR05] is able to reallocate BRAMs and multipliers and, unlike PARBIT, supports the VirtexII. pBITPOS [Kra06] is a similar tool operating in two modes: *simple* or *merge*. In *simple* mode, it can manipulate a full-height core (spanning entire FPGA frames). In *merge* mode, partial cores can be merged. FPGA equations to manipulate cores are well documented in [Kra06].

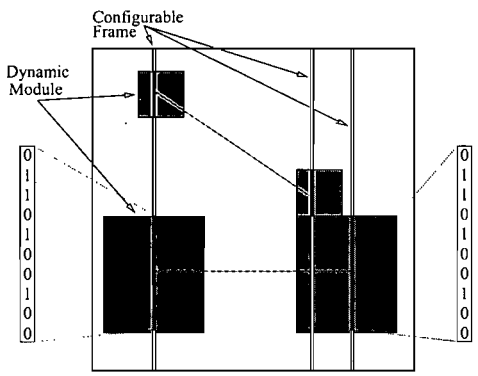


Fig. 2. Manipulation or cores (translation of two opaque cores)

For these tools, a core is a configuration (vector of bits) of rectangular region with a predefined and compatible interfac (usually interconnected with bus macros). The main limitation is that they manipulate *opaque* cores (black boxes) without

any regard to their content. They can extract cores from a full bitstream, translate and configure them. However, they are not able to merge overlapping cores or detect possible conflicts. This limitation comes from the fact that Xilinx does not document the content of the configuration frames and limits tools in their abilities to manipulate cores.

Reconfigurable Computing (RC) makes use of programmable logic (usually FPGA) and microprocessors to accelerate computations. RC architectures are of interest because they have been shown to speed up a wide range of applications [GK02]. Speed ups are obtained by dynamically reconfiguring the architecture to better fit the needs of the application. The RTR-JVM [SBA+06] (Runtime Reconfigurable Java Virtual Machine) is a platform that makes use of reconfigurable computing. The goal of this project is to automate reconfigurable computing for Java applications. The system uses a profiler to detect a set of *features* (contiguous segments of the algorithm). Selected features are translated to VHDL and synthesized by using standard Xilinx tools to produce a library of synthesized features. The virtual machine is able to dynamically load and unload features depending on the needs of the application. An important limitation is that the system is unable to produce new features on-the-fly.

Just-in-time compilation (JIT), also known as dynamic translation, converts code, at runtime, from a portable format (bytecode) to machine code. *JIT synthesis* is the concept of dynamically producing FPGA configurations from a core (net list) or higher level code. Currently, no system is able to perform this task. We believe it is possible by providing more detailed cores to a platform such as RTR-JVM. However, building such a system requires more information on the bitstream format to annotate the detailed cores. An annotation expresses an attribute of a core useful for deciding how and when to instantiate it (such as position of IO ports, resources, latency, ...)

This paper introduces a technique to determine the mapping between a configuration and its associated bitstream. We have used the proposed technique to find the mapping for the XC2VP2 FPGA (VirtexII Pro series). We demonstrate how this information is used to dynamically generate a configuration and its associated bitstream extremely fast using only our software running on the PowerPC embedded in the FPGA.

III. Logarithmic Reverse Mapping

Xilinx offers a tool (XDL) to transform a configuration into a textual yet proprietary format (NCD). This file can then be compiled using Bitgen to produce a bitstream. The problem we address consists in finding the positions of the configuration bits related to a given programming point, for all programming points. A linear analysis (one programming point per compilation) is not viable since the bitstream generation time is too long (minutes) and the programming points are too numerous (millions).

The idea behind *logarithmic reverse mapping* is to simultaneously resolve all programmable points by solving constraint sets. This way, the algorithm is asymptotically faster than other naive techniques and can be executed in a reasonable time.

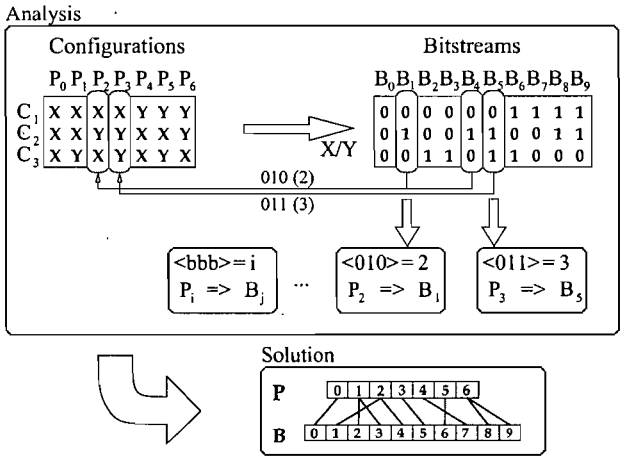


Fig. 3. Example of the Logarithmic Reverse Mapping on a 10-bit bitstream

The approach consists in producing a series of configurations $C_1 \ldots C_n$ where programming point settings evolve differently from each other. By observing the evolution of each bit in the bitstream, it is possible to resolve the mapping. Configuration points can take values the `X` or `Y` which are mapped to values `0` and `1` in the algorithm. For a set of programming points ($\{P_i\}$), we encode the value i in the sequence of configurations as illustrated in Figure 3. The matching bits B_j in the bitstream will then also be the binary encoding of i. Thus, bit sequence B_j encodes value i, its matching programming point.

The situation is more complex in reality because programming points can have more than two values, some constraints exist between programming points, some B_j are inverted or constant, etc. On FPGAs, different kinds of constraints can be found. There are *dependency* constraints when a programmable point can only be activated if one of its predecessors is activated. There are *configuration* constraints when programmable points depend on some configuration (voltage, I/O protocol, type, ...). There are *conflict* constraints when programmable points cannot be activated simultaneously because they share some resources. And finally, there are *sharing* constraints when two programmable points must have a related value (usually the same one) because their encoding share some bitstream bits. We illustrate these constraints in Figure 4.

We have been able to overcome these difficulties and to *reverse* the mapping in $O(\log|C|)$ time where $|C|$ is the number of programming points. This takes just a few days on a single workstation or a few hours on a cluster.

IV. Bitstream decompiler

To validate our approach, we have implemented a bitstream decompiler. This tool converts a bitstream into an XDL file. XDL files can be translated to NCD (Native Format Description) format and viewed in the FPGA Editor tool. As an experiment, we decompiled a 32-bit full-adder and observed using FPGA Editor that the original and decompiled designs were identical (Figure 5) (the inverse of the Bitgen tool).

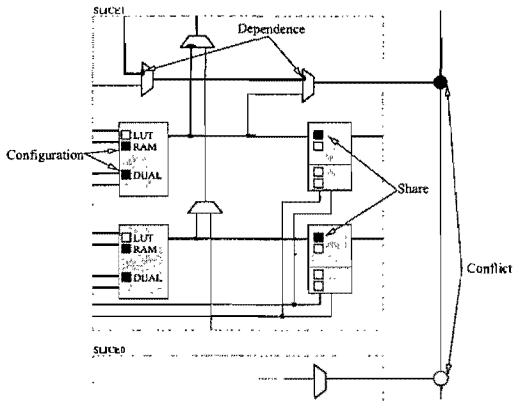


Fig. 4. 5 kind of constraints limiting bitstream generation

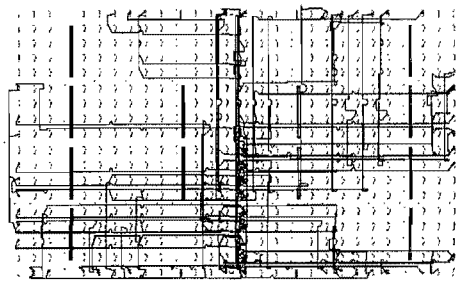


Fig. 5. Decompilation of a 32-bit full-adder on a XC2VP2 FPGA

This tool is useful for debugging dynamic designs. It is possible to suspend the FPGA, readback a configuration and import it in FPGA Editor. Readback can be done through JTAG or ICAP (Internal Configuration Access Port). This tool greatly simplifies the debugging of dynamic applications.

V. Annotated Tiles

Typically, the granularity of the components handled by RTR systems is the module (core). One of the major limitations is that modules cannot overlap. Moreover, interconnection (via macro bus) limits the width of the communication between modules. This granularity is not fine enough to realize a JIT that instead needs basic instructions (such as arithmetic operators, binary operators, multiplexers, ...). To solve this problem, we produced *annotated tiles*. The idea is to provide a set of fine-grained tiles annotated with information necessary to handle them correctly.

Figure 6 shows a pipeline built by merging basic tiles. To be able to produce this kind of module, tiles must be able to overlap and cannot pass through *bus macros* which is the design flow proposed by Xilinx [Xil04].

With information obtained by the technique of *reverse mapping*, we produce tiles (according to our specific needs) without using the Xilinx tools. Instead of representing a tile as a rectangular set of bits, our tiles contain a mask to specify which bits are really used. This mask can be used to merge overlapping tiles.

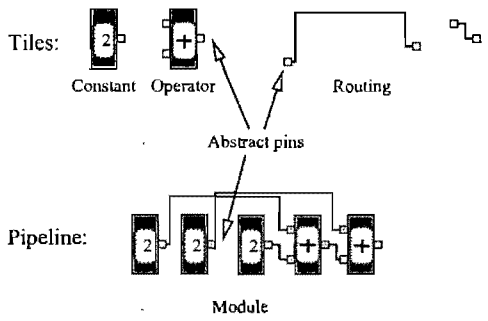


Fig. 6. Construction of a pipeline based on annotated tiles expression 2+2+2

With the aim of producing an instruction set for a prototype JIT, we determined some common properties of all tiles. The geometry of our tiles is constrained by the architecture of the FPGA. A CLB contains two columns of two SLICEs and each SLICE contains two LUTs producing two bits. As handling of the configurations by the JIT is made with 32-bit words and CLB configuration bits in a frame are 3 bytes high, we chose to produce tiles of 4 CLBs (Figure 7). Thus, we make 16-bit wide operators. Moreover, to allow the use of the carry chain it is necessary to use slices in column. As we observe in Figure 7, it is possible to put two operators in the same CLB, and they share the same switch matrix.

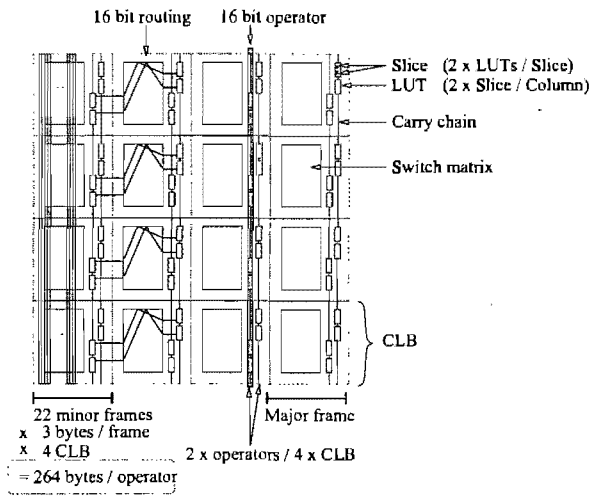


Fig. 7. Tile properties based on FPGA geometry

This organization makes the translation of tiles easy. Bits can be addressed using a relative address (represented by <BA, MJA, MNA, OFFSET, BIT>). The block addresses (BA) is zero for all CLBs. The major frame (MJA) is incremented for each column, from left to right. A column contains 22 minor frames (MNA). OFFSET represents the word of the frame and BIT specifies a bit in the word. Translating the configuration bits of a tile on the x-axis consists in adding a constant to the major frame (MJA), and translating on the y-axis consists in adding a constant to the offset (OFFSET). As ICAP uses relative addresses, we do not need to convert addresses to their absolute form. An operator tile needs 264 bytes to represent the needed configuration and the same for

its corresponding mask.

Abstract pins are used to represent a set of interconnexion points. For example, the LUT outputs of a column form a 16 bit value named L-O (left output) or R-O (right output) depending of the column parity (even = left, odd = right). In the same manner, the LUT inputs can be named L-I1, L-I2, L-I3, L-I4, R-I1, R-I2, R-I3, R-I4 because the LUTs take 4 inputs. Tiles produced by our tool are annotated with the abstract pins to specify their interface. Thus, a tile having a L-O output pin is compatible with a tile having an L-O input pin.

Tiles can be merged if and only if they do not use common resources. As there are too many resources to keep track of for tiles, we use *abstract resources* which represent a set of real resources. This leads to a more compact representation of resources without loss of generality. As an example, an operator that uses a LUT will probably use all other LUTs on the same column (16 LUTs). So, we produce the abstract resources L-LUTS and R-LUTS to represent the bundle of LUTs (left and right). This kind of abstraction makes sense because almost all tiles are symmetric for all CLBs (and often for all slices). This is also true for routing resources.

To minimize the number of tiles, we add the concept of *parameterizable tile*. As an example, tile FLUT4 represents a 4-input function and is parameterizable with a vector of 16 bits. The vector is the LUT configuration. Bitwise operators (AND, OR, XOR, NAND, ...) can all be implemented with this tile by passing the appropriate vector. We keep the mapping from the parameterizable vector to the configuration vectors with the tile.

Routing is handled similary to operators. A simple routing tile is an identity operator. For example, a tile with input L-O and output R-I1 is a routing tile. It is possible to implement some functions as routing tiles (such as shifting by a constant).

VI. Results

To go further toward our goal of on chip JIT synthesis, we synthesized a complete system on chip (SoC) with the XPS tool in the right half of a XC2VP30 FPGA on the ML310 board. The SoC is based on an embedded PowerPC processor, which is connected to several peripherals, in particular the ICAP. The design (processor and logic) run at 100 Mhz. Through the ICAP, we are able to produce small circuits on the fly in the left (initially empty) half of the FPGA (Figure 8). The selection of programming points, the production of the bitstream and the configuration are entirely done by the program running in the embedded PowerPC. We validated the circuits produced by using the READBACK function of the ICAP interface. Expressions up to 10 operators or constants take less than 200ms to synthesize and configure.

VII. Conclusion

The mapping between configuration points and bitstreams is a vital information for dynamic synthesis. We have proposed a logarithmic technique to determine this mapping in a realistic time. By using this information, we are able to produce sets of annotated tiles that can be used as basic configurable elements by other tools. The information obtained allowed us to implement the very first working prototype of a circuit implementing on chip JIT synthesis from a simple expression-based language.

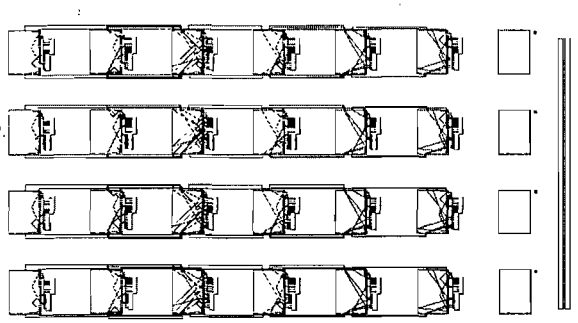


Fig. 8. JIT-Synthesis on a XC2VP4 of a simple expression

References

[GK02] Steven A. Guccione and Eric Keller. Gene Matching using JBits. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications*, pages 1168–1171. Springer-Verlag, Berlin, September 2002. Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications, FPL 2002. Lecture Notes in Computer Science 2438.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, September 1999.

[HL01] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University, July 2001.

[KJdlTR05] Yana E. Krasteva, Ana B. Jimeno, Eduardo de la Torre, and Teresa Riesgo. Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 77–83, Washington, DC, USA, 2005. IEEE Computer Society.

[Kra06] Krasteva. Virtex II FPGA Bitstream Manipulation: Application To Reconfiguration Control Systems. *Field Programmable Logic and Applications*, August 2006.

[SBA+06] Ron Sass, Parag Beeraka, Jason Agron, Jeff Young, David Andrews, Brian Greskamp, Srinivas Beeravolu, and Christian Trefftz. Run-Time Reconfigurable Java Virtual Machine on a Platform FPGA. *The Fourteenth Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[Xil04] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Technical Report XAPP290, Xilinx, September 2004.

[Xil05a] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Technical Report UG012, Xilinx, March 2005.

[Xil05b] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Technical Report DS083, Xilinx, June 2005.

Using Dynamic Reconfiguration to Implement High-Resolution Programmable Delays on an FPGA

Etienne Bergeron
Marc Feeley
DIRO, Université de Montréal

Marc-Andre Daigneault
Jean Pierre David
GRM, École Polytechnique de Montréal

***Abstract*—A digital circuit can be viewed as a network of transistors switching between low and high voltages. These transistors and the wires interconnecting them cause delays in signal propagation. In most cases, designers aim to minimize the delays in order to increase processing speed. Nevertheless, some applications such as delay lines, time to digital converters, asynchronous logic and others require the ability to precisely control a delay between two points in a circuit. This paper proposes a novel way to control the delays in an FPGA by dynamically configuring the routing matrices to build a path with the required delay and to calibrate the delays. Such low-level configuration is possible with a dynamic reconfiguration library we developed for Xilinx FPGAs. Our experiments on Virtex-II Pro devices show that any differential delay in a range of 947ps can be reached with a precision of +/- 18ps.**

I. Introduction

Field-Programmable Gate Array (FPGA) technology has matured a long way since its beginnings in the early 80's. Time-to-market is an increasingly important factor and, due to their high density, which is equivalent to millions of logical gates, FPGAs are gaining importance in many applications. Furthermore, as some devices allow for run-time reconfiguration, new concepts such as virtual hardware have emerged and much research has been conducted in the area of dynamically configurable hardware. These concepts could be the premises of the next computing revolution since manufacturers such as Intel, AMD and HP seem to consider using them or interfacing to them. However, modern work in this area faces a critical problem: the lack of tools.

Over the years, several research projects in the area of dynamic reconfiguration have used the JBits SDK [GLS99] for the Virtex FPGAs manufactured by Xilinx. One idea behind using a library such as JBits is to get a finer granularity level in the design process and to reduce the limitations imposed by Xilinx's standard design flow. But as the technology kept progressing with the Virtex-II Pro, Virtex-4 and Virtex-5 families (all offering the capability of run-time reconfiguration), no complete tool or library such as JBits has been made available to designers and researchers to keep up with theses advances.

In a previous work, we have been able to understand the mapping between an FPGA bitstream and the programming points in Xilinx FPGAs [BFD07]. Since then, other researchers have published similar information [NR08]. Xilinx is also keenly interested in developing tools in this field [LBM+06]. Undoubtedly, low-level dynamic configuration will take an important place in the near future. In this paper, we present a new advance towards bridging the gap between the low-level dynamical configuration possibilities offered by modern Virtex FPGAs and current development tools. We developed a compact C library and related tool set for low-level dynamic reconfiguration, which currently supports Virtex-II Pro and Spartan-3 devices. In order to illustrate some of the new possibilities offered by this library, we present an application to delay management in a Virtex-II Pro FPGA.

Some applications require precise and known delays. This issue has already been addressed for clock signals by using programmable phase lock loops which change the phase of a clock signal. But this approach relies on the cyclic nature of clocks. Applications such as Time to Digital Converters (TDC), delay lines and asynchronous logic require the ability to measure and possibly modify the delay between two points in a circuit. These applications have always been confronted to a range/resolution dilemma and seldom FPGA implementations are mentioned in the literature [Kho06]. Indeed, even if current FPGAs offer many degrees of freedom to adjust the delays, proprietary tools are designed to optimize them, i.e. to minimize the delay of selected (critical) paths.

From a more general point of view, some applications require to control the configuration of an FPGA at the finest granularity, which is currently impossible with the standard design flow. The tools and methodology we developed offer such a fine control and the ability to modify the design's configuration at run-time. The experiments conducted on a Virtex-II Pro FPGA demonstrate that with dynamic reconfiguration it is possible and easy to have a very fine control on the delays by taking advantage of the architecture of the switch matrices. Our approach, which allows very precise programmable delay circuits, could lead to full FPGA integration replacing external dedicated devices.

Section 2 presents the library, which is part of our dynamic reconfiguration framework. Sections 3 and 4 explain the presence of delays in FPGAs and propose a way to control them. Section 5 gives the implementation details concerning delay measurement and calibration, and gives experimental results.

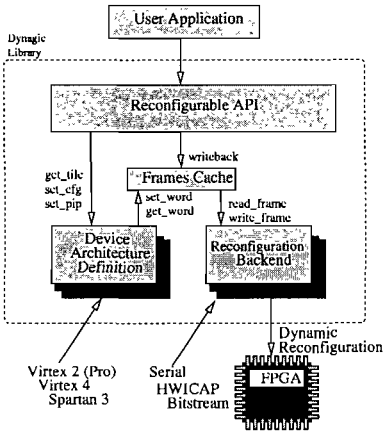


Figure 1. Dynamic reconfiguration framework

II. Dynamic Reconfiguration Framework

The DYNAGIC group [BDFD] is implementing development tools for dynamic applications. A tool of central importance is the `dynagic` library which abstracts the reconfiguration mechanism of Xilinx FPGAs. The purpose of this library is to allow an alternative design flow to the one offered by Xilinx [Xil04]. The main limitation of the Xilinx's flow is that it does not allow dynamic generation of modules; all modules must be statically generated.

Our library supports on-the-fly construction of fine-grained modules at the same level of abstraction as FPGA Editor. The developer can activate/deactivate each programmable point by using a C API.

As the design in Figure 1 shows, the library is divided so that it is possible to plug specific modules that abstract parts of the required functionality.

The *device architecture definition* provides functionalities related to a specific FPGA family (e.g. Virtex-2, Virtex-2 Pro, Spartan-3, Virtex-4...). The *reconfiguration backend* provides raw access to the FPGA configuration memory. As dynamic applications may run in different contexts (embedded or on a host), the library has different implementations. The *frame cache* provides faster reconfiguration performance by caching read/write operations. Thus, by using the reconfiguration API, it is possible to construct a dynamic application with its specific needs.

Figure 2 shows an example of using the library. The `dg_open` function initializes the library. The *reconfiguration backend* (ICAP, JTAG, serial port, ...) is dynamically selected depending on the first program argument (`argv[1]`). Thus it allows the same program to run on a host PC or on the embedded processor. The `dg_get_tile` function returns a handle of the tile at a specific location of the device grid. The `dg_get_site` function returns a handle to a site in a specific tile. The `dg_set_cfg` sets the value of a programmable point at a specific site. The `dg_set_pip` function activates a programmable interconnect point (PIP) in the switch matrix of a specific tile.

```
#include "dynagic.h"

#define LUT_NOT_G2 ~0x3333
#define LUT_F3     ~0xF0F0

int main(int argc, char** argv) {
  dg_system_t sys;
  dg_tile_t tile;
  dg_site_t slice;

  dg_open(&sys,argv[1]);
  dg_capture(&sys);

  tile  = dg_get_tile(&sys, 1, 1);
  slice = dg_get_site(&sys, tile, V2_COMP_SLICE1);

  dg_arch_set_lut(slice, V2_RESS_G,     LUT_NOT_G2);
  dg_arch_set_cfg(slice, V2_RESS_GYMUX, V2_VAL_G);
  dg_arch_set_cfg(slice, V2_RESS_YUSED, V2_VAL_0);
  dg_arch_set_lut(slice, V2_RESS_F,     LUT_F3);
  dg_arch_set_cfg(slice, V2_RESS_FXMUX, V2_VAL_F);
  dg_arch_set_cfg(slice, V2_RESS_XUSED, V2_VAL_0);

  dg_arch_set_pip(tile, V2_WIRE_Y1, V2_WIRE_DY1);

  dg_frames_cache_writeback(&sys);
  dg_close(&sys);
}
```

Figure 2. `dynagic` library example

III. Design considerations for the delay circuit

A digital signal is transmitted from a source component to a target component by routing it through a chain of intermediate components. Each one propagates the signal to the next component in the chain until the target is reached. The FPGA's configuration determines which route the signal will take. Each component on a route contributes to the total delay from source to target. The delay depends on wire length and capacitance, transistor switching speed, temperature, and other factors.

On a Xilinx Virtex FPGA, the intermediate components are either wires, logic elements, or switch matrices (which probably contain wires, pass transistors and buffers). A common approach of implementing a delay on Virtex devices is to route the signal through a chain of N look-up tables (LUTs). The LUTs have a propagation delay of roughly 250ps, so the total delay is roughly $N \times 250$ps plus the delay of the routes needed to interconnect the LUTs in a chain and connect them to the source and destination. Each connection typically adds a delay in the hundreds of picoseconds, but sometimes more than a nanosecond. Some tools, FPGA Editor in particular, can give a maximum delay for these connections but not the actual or expected delay. So LUT-based delays suffer from a rather coarse resolution, and a substantial uncertainty on the actual delay.

The approach we developed uses only the switch matrix to implement the delay. The switch matrix is a component which can be configured to transmit a signal from one of several input pins to one (or more) of its output pins. The pins of the logic elements and the inter-tile network wires are connected to these inputs and outputs. For a given pair of input and output pins there is typically a very large number of ways to route the

signal within the switch matrix. This is due to the architecture of the switch matrix which can route a signal from an input pin to an output pin unconnected externally, and *bounce* the signal to another output pin (see Figure 3).

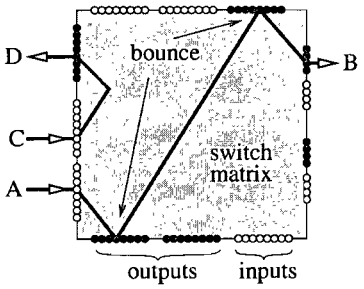


Figure 3. A switch matrix configured with the routes $A \rightarrow B$ and $C \rightarrow D$ of depth 3 and 1 respectively

We define the *depth* of a route as the number of switch matrix wire segments that are used. The depth is also the number of PIPs that must be activated in the configuration to create this route. The total delay of a route depends on the choice of wire segments and is roughly proportional to its depth. There is a considerable disparity in the delay contributed by the switch matrix wire segments. Given the high number of possible routes, many different delays can be obtained. A table of routes up to a certain depth and the associated delay could be created and used by a tool to achieve, between an input and output pin, a delay as close as possible to the desired delay in a certain time range.

The actual delay of a route is influenced by device properties which can vary from part to part, and by environmental parameters such as temperature which can vary over time. To achieve repeatability and accuracy it is desirable to calibrate the device during its operation. To reduce the drift over time caused by temperature changes the delay for the set of routes in the table must be measured repeatedly throughout the device's operation. To do this we use dynamic reconfiguration.

IV. Switch Matrix Routes

The switch matrix routes used by our method are intentionally non-optimal, so they cannot be generated with standard tools. The set of pins and PIPs contained in a switch matrix are described in the device's XDL report generated with the command "`xdl -report -pips ...`". We have used that information and a simple depth first search to create a database of all the possible routes from inputs to outputs up to a depth of 9.

To test our approach we have generated the routes between the output of LUT G in SLICE1 of a CLB (pin Y), and the inputs F1, F2, F3 and F4 of LUT F in the same slice. There are 546 different routes with a depth ≤ 9. Other pins could have been used, but these were chosen because they are easy to connect to other components. The signal to delay must be routed to one of the inputs of LUT G and the delayed signal will be on the output of LUT F (pin X). The switch matrix must be configured to implement the appropriate route for the desired delay, LUT F must be configured to propagate tl output of the switch matrix to its output pin X, and LUT G must be configured to propagate the appropriate input to the output pin Y. This approach allows the total delay to be adjusted without changing the routing of the signal to LUT G and from LUT F (see Figure 4). Note that the routing to LUT G and from LUT F can be avoided in some cases by using these LUTs to perform useful operations. Avoiding the LUTs altogether is also possible, but it increases the difficulty of delay measurement and calibration.

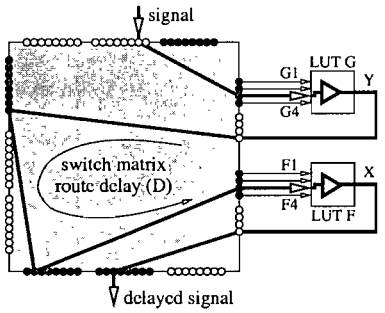


Figure 4. Configuration of the switch matrix and LUTs when a signal is being delayed

V. Delay Measurement and Calibration

The switch matrix delay for a given route can be measured with a ring oscillator and a ripple binary counter driven by the oscillator (see Figure 5). A ring oscillator could be built by configuring LUT G as an inverter and LUT F as a buffer, and sending the output from pin X to an input of LUT G. The signal at pin X is also sent to the ripple counter. This oscillator might exceed the maximal switching frequency of the FPGA so to be safe we added 2 other buffers in the feedback loop to slow down the oscillator to under 400MHz. Given an accurate time base, such as an external crystal, it is a simple matter to read the counter after a given route has been configured, and to read it again after T seconds. The difference X is related to the period P of the oscillator by the formula: $P = T/X$. The half period of the oscillator is equal to the switch matrix routing delay D plus R, the propagation delay of the LUTs and the routing to connect them. So $D = P/2 - R$. It must be observed that, while R is an unknown, it is unaffected by the switch matrix routing. This means that the delay difference between two routes can be obtained. This is often the most important information (an absolute time is rather useless when the routing delay of the incoming signal can't be known precisely). If needed R can be approximated by choosing a route with a delay that is known to be short (using the maximal delay reported by FPGA Editor). If we use the shortest route (Y1 $\rightarrow$ F3_B1 $\rightarrow$ F3_B_PINWIRE1), which has a reported maximal delay of 53ps, the half period measured is 1364p so R = 1338ps +/- 27ps.

During calibration, the tile containing the delay line and a few other CLBs are reconfigured dynamically to implement

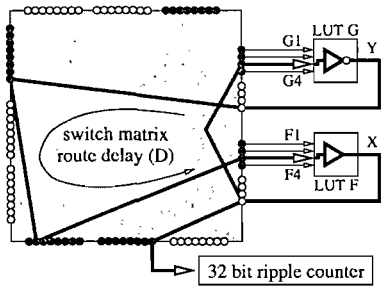


Figure 5. The configuration of the switch matrix and LUTs when the route's delay is being measured

the ring oscillator and ripple counter. The value of the counter is accessed by the processor through the ICAP. It takes less than 10 seconds to measure the half period of over 500 routes. This could be made faster using a dynamically reconfigured dedicated control circuit. To verify the stability of the delays we performed the measurements 20 times. The measured half period for a given route varied by at most 1ps. We also performed a linear regression to try to explain the half period as a linear function of the maximal delay (M) reported by FPGA Editor. The regression gives $P/2 = 1317ps + 0.598 \times M$. Figure 6, which plots the residual errors, shows that there is a good correlation between M and D.

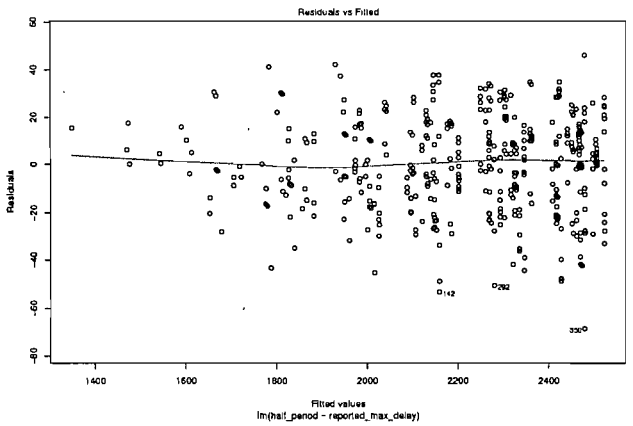


Figure 6. Residual errors for $P/2 = 1317ps + 0.598 \times M$

Table I gives the switch matrix route delay (D) obtained by substracting 1338ps (R) from the half period measured for the routes. To save space, out of the 546 routes measured, only the 10 shortest delay routes and the 2 longest are shown. Switch matrix route delays between 266ps and 1213ps, a range of 947ps, are spaced at most 36ps apart. This means that for any target delay in this interval a route exists whose delay is no more than 18ps away from the target. The precision increases substantially if we restrict the delay to a subinterval in the larger delays. For instance, between 708ps and 1213ps, a range of 505ps, the error is +/- 8ps. This higher precision is due to the increasing density of route delays for longer routes.

26ps	Y1 F3_B1 F3_B_PINWIRE1
138ps	Y1 W2BEG4 F3_B1 F3_B_PINWIRE1
138ps	Y1 S2BEG4 F3_B1 F3_B_PINWIRE1
153ps	Y1 OMUX6 F3_B1 F3_B_PINWIRE1
207ps	Y1 S2BEG2 F1_B1 F1_B_PINWIRE1
208ps	Y1 E2BEG1 F1_B1 F1_B_PINWIRE1
266ps	Y1 OMUX6 W2BEG2 F3_B1 F3_B_PINWIRE1
268ps	Y1 OMUX4 W2BEG2 F3_B1 F3_B_PINWIRE1
273ps	Y1 OMUX15 E2BEG9 F4_B1 F4_B_PINWIRE1
279ps	Y1 OMUX15 N2BEG9 F4_B1 F4_B_PINWIRE1
...	
1209ps	Y1 E2BEG1 BX0 BY3 BX3 BY2 BX1 BY1 F1_B1 F1_B_PINWIRE1
1213ps	Y1 S2BEG2 BX0 BY3 BX3 BY2 BX1 BY1 F1_B1 F1_B_PINWIRE1

Table I
MEASURED SWITCH MATRIX ROUTE DELAYS FOR ROUTES BETWEEN Y AND F1, F2, F3 AND F4.

The number of routes increases exponentially with the route depth, whereas the route delay increases roughly linearly with the route depth.

VI. CONCLUSION

Switch matrices in Virtex FPGAs allow numerous ways to route a signal for a given pair of input and output points. Standard tools automatically configure these matrices to attempt to get the minimal delay for a given set of routing constraints. We have presented a dynamic reconfiguration library with a flexible and efficient API allowing the dynamic configuration of an FPGA at the finest level of granularity. Thanks to this library, we have been able to generate custom routes in the switch matrices and measure their delays. This information was used to implement a way of controlling the delays in an FPGA with a precision varying from 8ps to 18ps, depending on the required delay range. This work gives a taste of the previously unimplementable applications now achievable with our library.

REFERENCES

[BDFD] Etienne Bergeron, Marc Andre Daigneault, Marc Feeley, and Jean Pierre David. Dynagic Web Page. http://www.dynagic.org.

[BFD07] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Toward On-Chip JIT Synthesis on Xilinx Virtex-II Pro FPGAs. In *50th International Midwest Symposium on Circuits and Systems/5th International Northeast Workshop on Circuits (MWCAS/NEWCAS), Montreal, Canada*, August 2007.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing, 1999.

[Kho06] Amir Mohammad Amiri; Mounir Boukadoum; Abdelhakim Khouas. Low Dead Time, Multi-hit FPGA-Based Time-to-Digital Converter. *Circuits and Systems, 2006 IEEE North-East Workshop on*, pages 29–32, June 2006.

[LBM+06] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young, and Brendan Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *FPL*, pages 1–6, 2006.

[NR08] Jean-Baptiste Note and Eric Rannaud. From the Bitstream to the Netlist. In *Sixteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2008.

[Xil04] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Technical report, Xilinx, September 2004.

5.1 Conclusion

La reconfiguration dynamique sur les FPGAs de Xilinx est jeune et prometteuse mais manque gravement d'outils. Nos travaux de recherches ont permis de combler partiellement ce manque et de permettre d'essayer de nouvelles applications qui ne peuvent être envisagées avec les techniques classiques de développement.

Les outils développés tels que *genbit* qui décompile une région du FPGA est le seul outil disponible pour analyser le résultat produit par nos générateurs de configurations et ainsi déboguer les applications. Sans ces outils, il est fastidieux de construire de telles applications.

Nous croyons que de nombreux outils sont réalisables suite à la construction de notre bibliothèque : un simulateur de FPGA supportant la reconfiguration dynamique ou encore un générateur de *bus macro*.

Chapitre 6

Compilateur dynamique

Dans ce chapitre nous expliquerons l'évolution de nos techniques de synthèse rapide ainsi que les différentes versions du compilateur réalisées afin de parvenir à la version présente de notre compilateur : Dynabit.

Un article, présenté à la conférence *Compiler Conference 2008*, résume Dynabit [BFD08], notre premier compilateur effectuant de la synthèse à la volée efficacement. Par la suite, nous expliquons les améliorations apportées à ce compilateur pour obtenir la version récente de Dynabit.

6.1 Évolution

Le progrès réalisé par la conception de la bibliothèque (présentée au Chapitre 5) était ce qu'il manquait pour permettre la synthèse dynamique. La partie caudale de notre compilateur repose sur notre bibliothèque. À partir de ce point, plusieurs versions du compilateur ont vu le jour dans le but d'essayer différentes approches et d'améliorer les performances pour finalement obtenir un résultat qui satisfait les contraintes de temps et d'espace mémoire.

Les premières versions du compilateur dynamique étaient rudimentaires et ne pouvaient obtenir des performances raisonnables. Par exemple, le premier algorithme de placement et routage en était basé sur le retour arrière (*backtrack*) qui tentait de connecter des tuiles abstraites.

Celui-ci a été présenté à la conférence *NEWCAS 2007* [BFD07]. Quoique fonctionnel, il ne pouvait remplir le contrat de performance requis pour la synthèse à la volée puisque le temps de synthèse d'expressions simples (environ 2 ou 3 additionneurs) est de l'ordre de 120 ms mais augmente exponentiellement (plusieurs secondes) en fonction de la taille de l'expression.

Le concept de tuile abstraite a alors été écarté pour faire place à des algorithmes plus classiques. Un algorithme basé sur le recuit simulé a été utilisé pour le placement et l'algorithme du chemin le plus court de Dijkstra a été utilisé pour le routage. L'implantation de ces algorithmes nous rapproche des techniques utilisées par le projet Warp, ce qui permet de réduire substantiellement le temps de compilation.

Tout de même, les performances n'étaient pas celles que nous visions. Ainsi, des heuristiques efficaces ont été introduites pour effectuer la plus grande partie de la tâche rapidement et les anciens algorithmes sont utilisés comme alternatives lors d'échec des algorithmes plus simples. À ce stade, nous avons un compilateur relativement efficace que nous avons nommé Dynabit, présenté dans ce chapitre par notre article publié à *Compiler Conference 2008* [BFD08].

Par la suite, nous avaons amélioré le routage spécifiquement pour le VirtexII-Pro suite à l'analyse manuelle des matrices de routage et des observations ainsi réalisées. Les modifications apportées aux algorithmes permettent à Dynabit d'obtenir des temps de compilation raisonnables pour être utilisés comme un JIT. Ses performances sont détaillées au Chapitre 7.

Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs

Etienne Bergeron, Marc Feeley, Jean Pierre David

DIRO, Université de Montréal
GRM, École Polytechnique de Montréal

Abstract. JIT compilation is a model of execution which translates at run time critical parts of the program to a low level representation. Typically a JIT compiler produces machine code from an intermediate bytecode representation. This paper considers a *hardware JIT compiler* targeting FPGAs, which are digital circuits configurable as needed to implement application specific circuits. Recent FPGAs in the Xilinx Virtex family are particularly attractive for hardware JIT because they are reconfigurable at run time, they contain both CPUs and reconfigurable logic, and their architecture strikes a balance of features.
In this paper we discuss the design of a hardware architecture and compiler able to dynamically enhance the instruction set with hardware specialized instructions. A prototype system based on the Xilinx Virtex family supporting hardware JIT compilation is described and evaluated.

1 Introduction

Software just-in-time (JIT) compilation is a well-known technique for improving the execution speed of virtual machine interpreters. The virtual machine identifies through run-time profiling which program parts are critical to its performance (so called *hot spots*) and compiles these parts into optimized machine code that is directly executed by the processor. Because the program's behavior evolves throughout its execution, the virtual machine continually monitors the program to find the new hot spots to compile. The compiled code is often stored in a limited size cache which contains the most recently compiled parts of the program. Speed-ups are obtained when the monitoring and compilation effort are more than compensated by the execution time savings of the compiled hot spots. For this reason, time-consuming complex optimizations are typically not performed by JIT compilers.

Hardware just-in-time compilation is an extension of this model to field-programmable gate arrays (FPGAs). Figure 1 contrasts the software and hardware JIT models. FPGAs are highly parallel configurable digital circuits whose behavior can be tailored to a specific application in the field through the process of *configuration*. Normally this configuration is performed at power-up, but some FPGA families, namely the Xilinx Virtex II [19] and above, can reconfigure sections of the device at run time. This FPGA family also supports up

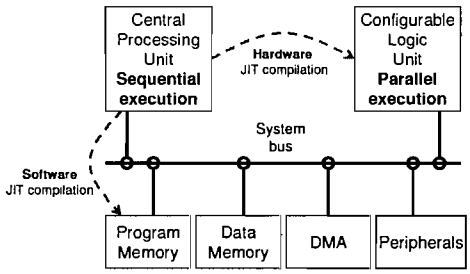


Fig. 1. Software and hardware JIT models

to two classical processors inside the reconfigurable logic, which can be used to implement a virtual machine interpreter in software. A hardware JIT compiler, embedded in the virtual machine interpreter, will compile program hot spots into the low-level description (*bitstream*) of a digital circuit performing the same computation. The circuit's layout and position in the reconfigurable logic as well as its interconnection with the processor running the virtual machine interpreter are determined dynamically.

Circuit synthesis, placement and routing are rather time-consuming tasks. It is not uncommon for standard synthesis tools to take several minutes on a high-performance workstation to produce the bitstream for a simple computation. This high cost of compilation must be amortized on abnormally long running hot spots to achieve any speed-up. In order for hardware JIT compilation to be useful for executing more typical programs, it is necessary to decrease the compilation time by a few orders of magnitude. This is the obstacle we tackle in this work.

Since the bitstream format of dynamically configurable FPGAs is not documented by the vendors, all the related work involving dynamical configuration are based on vendor-supplied proprietary compilation tools. Our recent work [1] has enabled us to extract enough information on the bitstream format to be able to generate partial bitstreams on-the-fly in a fraction of a second without any proprietary tool. We believe this is a key result on the road to general purpose reconfigurable computing.

This paper describes the fast synthesis technique we have designed for a hardware JIT compiler. Sections 2 and 3 report on related work and give some background information on FPGAs. We describe a prototype compiler in Section 4. An evaluation of its performance is given in Section 5. We specifically focus our attention on the synthesis times.

2 Related Work

Reconfigurable architectures such as PipeRench [8,16] and WASMII [15] gave birth to the concept of *virtual hardware*. The idea is analogous to virtual memory.

Since the hardware resources on a chip are limited, it may be interesting to "store hardware" out of the chip and "swap hardware" when required. Such hardware manipulations require a dedicated area of the chip that is configurable. This means that it is possible to alter the logic or the connections via software. The store and swap mechanisms thus actually access the configurations bits, which are also called *bitstream*.

Xilinx Virtex II and above FPGAs, which support dynamic reconfiguration, have also been used for virtual hardware implementation [4]. Two synthesis flows are proposed by Xilinx to handle the creation of dynamic modules using their tools [18]. Essentially, each possible global configuration of the FPGA must be pre-compiled using the standard synthesis process. Then, partial bitstreams are extracted from the complete bitstream for each module. The swapping from one configuration to another is achieved by sending a partial configuration to the FPGA. Standard tools were not originally developed for this type of compilation and their use imposes severe limitations on the design of virtual hardware. At present, it is still very difficult to develop, debug and guarantee the stability of dynamic applications.

The RTR-JVM (Reconfigurable Run-Time Java Virtual Machine) [9] proposes a different approach where the concept of dynamic configuration is integrated in the language and its virtual machine. This architecture allows the dynamic loading of hardware modules produced from Java source code and translated to VHDL code. By profiling the hot spots, the system is able to identify good candidates for hardware implementation. The main limitation of this system is that modules are produced statically (by the standard synthesis flow) and preliminary executions are required.

Warp processors [11] are another example of hardware virtualization. The authors propose to translate binary code to hardware [17] in order to make the production of dynamic modules transparent to the user. The advantage of this mode of execution is that it can be integrated to a conventional processor. That work is based on a custom FPGA [12] as well as custom tools, compilers and algorithms [13,14], which benefit from the regular structure of the custom FPGA. A drawback of custom FPGAs is that they lag commercial ones in terms of size, speed, and cost. The Warp project nevertheless demonstrated that JIT compilation is viable on modern FPGAs.

We believe that on-the-fly generation of dynamic modules is appealing and needs more investigation. In this paper, we demonstrate that these techniques are applicable to commercial FPGAs despite their limitations in terms of architecture and synthesis tools.

3 Target Architecture

Our methodology assumes a dynamically reconfigurable FPGA with an embedded processor as illustrated in Figure 2. The application and the compiler are stored in an external memory and run on the embedded processor. Some peripherals can be mapped inside the FPGA configurable logic, allowing the

application to perform input/output. This zone is static, which means that it is configured at power-up and will not be modified later. The rest of the logic constitutes the *reconfigurable zone*, which will behave as a hardware cache for the dynamically generated specialized instructions. The compiler produces these instructions on-the-fly and configures the dynamic zone to make them available to the application through standard I/O instructions.

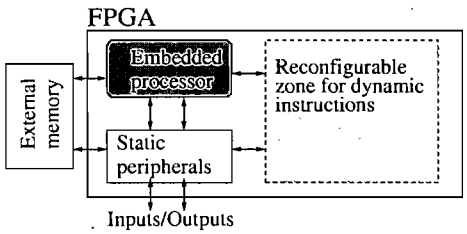


Fig. 2. Diagram of a system supporting dynamic reconfiguration on a FPGA

In practice, we use Xilinx Virtex-II Pro devices, which are one of the few commercial devices currently supporting dynamic and partial reconfiguration. Moreover, these devices (and later series) are the most dense FPGAs available on the market. Figure 3 shows the layout of a small Virtex-II Pro device with one embedded PowerPC processor.

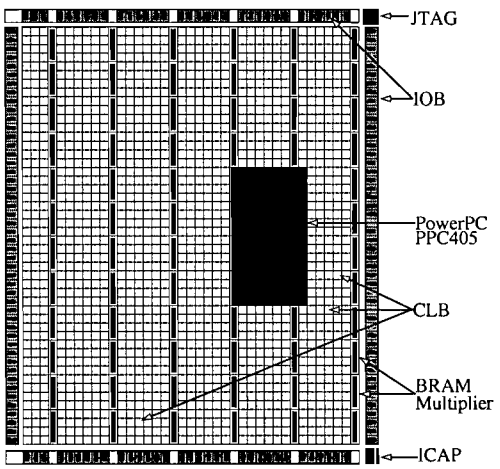


Fig. 3. Virtex-II Pro Device Grid

A FPGA is mostly a grid of configurable blocks that can be interconnected in a configurable way. Input/Output Blocks (IOBs) are physically connected to the external pads. They are placed on the grid's periphery. Configurable Logic Blocks (CLBs) are the heart of the processing power of the FPGA. Basically, each CLB contains user-defined lookup tables (LUT), registers and a programmable routing matrix to manage the connections to other CLBs. Virtex-II Pro FPGAs also contain hard-wired cores to increase the design density. The densest FPGA (2VP100) is equipped with 2 embedded PowerPC processors running at 400 MHz, 12 clock management devices, 444 18x18 bit multipliers, 444 block RAMs (18 kbits each) and almost 100000 CLBs spread all over the chip.

Some special blocks are located at the chip's periphery. The JTAG block is a serial interface provided for low level debugging. The Internal Configuration Access Port (ICAP) is used to configure the FPGA. In our architecture, it is connected statically to the PowerPC processor to enable dynamic configuration through standard I/O instructions.

Given this architecture, the challenge we face is to quickly generate the configuration of the dynamic zone to enable efficient JIT hardware. More precisely, we have to configure the blocks and their interconnection. This is addressed in the next section.

4 Compiler Architecture

The next step is to build an execution environment running on the processors. Instead of writing a whole system from scratch, we decided to port the Gambit [6] Scheme to C compiler to the embedded PowerPC. This way, we inherit all Scheme features and a dynamic execution model (dynamic software module loading, dynamic code interpretation, etc). Gambit uses a fast interpretation technique [7] and gives access to the AST representation of the running program which the JIT compiler uses.

Many languages can benefit from this kind of execution and we believe they can use a common back-end. High-level language synthesis, which consists in choosing how high-level concepts are mapped to hardware, is the most language-dependent phase but the running time of this phase is not a bottleneck. We have already addressed this issue in a previous work in the context of a static hardware compiler [2].

The hardware JIT compilation process follows the standard phases used to produce a FPGA bitstream: synthesis, technology mapping, place and route, bitstream generation and configuration. The synthesis phase translates a Scheme expression into a representation that explicitly indicates how high-level concepts are implemented (e.g. pipeline, state-machine,...) to take advantage of hardware. The technology mapping phase attempts to map components generated by the synthesis to patterns and resources existing on the FPGA (multipliers, block RAMs, slices,...). The place and route phase finds a location for each of the resources and connects them by assigning appropriate wires. Finally, the

bitstream generation produces a partial bitstream that will be downloaded into the configuration memory in the configuration phase.

In a static compiler, these phases are optimized to produce a high-density, fast and low-power design. Problems faced by a JIT compiler are not the same and new techniques must be developed; the compilation time is one of the most important aspects. To minimize it, some trade-offs must be made. Thus, we prefer greedy algorithms over computation-intensive algorithms that could yield a more optimized design.

In the following sections, we describe the algorithms and design choices made at each phase of the compilation.

4.1 Source Language and High-Level Synthesis

High-level synthesis of digital systems consists in transforming a behavioral (algorithmic) description of a design into a RTL (register transfer level) description of the design. This phase determines how the high-level concepts of the programming language are translated to low-level primitives.

Decisions must be taken on how high-level concepts can be implemented in hardware. For example: should a function be implemented as a pipeline able to handle parallel calls, or as a state machine, much more compact but unable to handle parallelism? Other decisions must be taken such as which communication protocol to use between software and hardware.

We chose to use Scheme [10] because it is a dynamic language quite easy to use and learn. Scheme provides a small set of primitive constructs with which most high-level features can be implemented. This simplifies the structure of the compiler.

To elegantly provide dynamic compilation to hardware, our compiler is made available as a user-callable `synthesize` primitive. This primitive maps a function object, possibly a closure, onto the reconfigurable hardware. The sole parameter is a function and it returns a function which performs the same computation in hardware (i.e. semantically `synthesize` is the identity function). If the compilation fails, the primitive returns the argument unchanged. Thus, the complexity of hardware synthesis is hidden behind a single function.

This approach meshes nicely with *function closures* which remember their environment of definition. For people versed in functional programming languages dynamic hardware synthesis with this extension is very natural. Hardware specialization is simply viewed as the partial evaluation of the function body given the binding of variables in the definition environment (i.e. the closure's free variables).

Figure 4 shows a use of the `synthesize` primitive. The call `(adder 4)` returns the closure which is an instance of the lambda-expression at line 3. This closure is a specialized version of an adder which always adds 4 to its argument. By passing this closure to the `synthesize` primitive, the system dynamically compiles the closure, configures the dynamic zone and returns another closure able to communicate with the dynamic instruction. When `map` calls that closure,

```
1 (define adder
2   (lambda (x)
3     (lambda (y)
4       (+ x y))))
5
6 (pretty-print
7   (map (synthesize (adder 4))
8        '(1 2 3 4 5 6 7 8 9 10))
```

Fig. 4. Hardware synthesize/invocation of a specialized instruction

the argument is sent to the hardware's input and the result is obtained from the hardware's output.

High-level synthesis is complex because many issues must be taken into account (e.g. memory access, global variables, continuations, threads...). In our current prototype, we only support simple expressions containing arithmetic and logic operators. Much work remains to be done to determine how to perform high-level synthesis on general purpose languages.

4.2 Technology Mapping

Technology mapping consists in transforming technology-independent logical circuits into a technology-dependent mapping on a given technology. Typically, this phase is driven by a set of technology patterns defined in a library. Mapping is constrained by characteristics such as available physical gates, delays, available power and area.

Compiler back-ends solve a similar problem when generating code. Two approaches are typically used: top-down and bottom-up. The top-down approach, often called Maximal Munch, consists in finding, in the library, a pattern that matches as much as possible from the root of the tree. Parts that were not matched are processed in the same way. Another approach is to use dynamic programming to find a way to cover the tree with a set of patterns.

Our pattern library is designed to be a bridge between a high-level language and the low-level requirements imposed by the FPGA fabric. A typical operator has a height of 4 CLBs and uses one slice column (i.e. half of a CLB column). The use of a column of slices can be justified by the way fast carry chain logic works. We decided to use 4 CLBs (and 16-bit operators) because the configuration process works with 32-bit words and a CLB is 3 bytes wide. Thus, we obtain operators representable by a sequence of three 32-bit words. In addition, embedded multipliers are 18x18-bit and can implement 16-bit multiplication directly. Figure 5 shows typical operator patterns of our library.

In our prototype, we decided to use a fast top-down approach. This phase takes less than a millisecond to perform mapping of a specialized instruction. It does not require much memory and is far from being the bottleneck of synthesis.

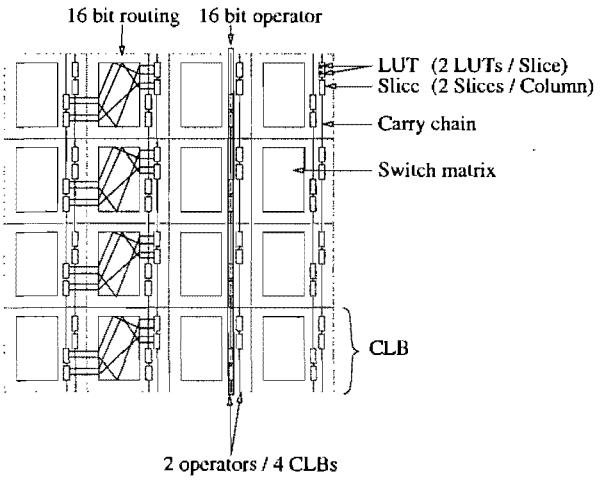


Fig. 5. Mapping of operators onto the FPGA fabric

4.3 Place and Route

Placement consists in finding a location for each component of the design. Locations are chosen to minimize the distance between dependent components and to maximize the probability of success of the routing phase. Sometimes, other criteria such as power consumption must also be taken into account. Routing consists in finding a path through static wires for each net.

Placement and routing are the slowest phases of synthesis on FPGA. Running times of several minutes are common for these phases. Although phases are run separately, they are interdependent. A good placement facilitates routing, whereas a bad placement makes routing extremely complex. Because of the complexity of these phases, their algorithms are crucial to be optimized if we want to attain short compilation times. To understand current fast techniques, we briefly explain the VPR and the ROCR tools which are the fastest available algorithms for FPGA compilation. We then describe algorithms accelerating these phases.

The VPR (Versatile Place and Route) [3] tool uses the simulated annealing algorithm for placement. The basic idea is to perform a random initial placement of all components. At each iteration, components are randomly swapped with a probability function of temperature and cost. The temperature gradually decreases until a threshold is reached. The VPR routing algorithm is an improvement over the Pathfinder negotiated algorithm [5]. Initially, it routes all nets with the shortest path regardless of the availability of resources. Dijkstra's algorithm is used to find the shortest path. At each iteration, every route is sequentially re-routed by the lowest cost path. The cost of using a resource is a function of the overuse of that resource. At the end of the iteration, the costs of routing resources are adjusted accordingly to the amount of overuse in the previous iteration. By gradually increasing the cost of oversubscribed routing

resources, the algorithm forces nets to avoid them and to use alternative routes. Although VPR is quite fast, it is not fast enough for JIT place and route.

ROCR (Riverside On-Chip Router) [13] is designed for hardware JIT compilation. It uses the basic cost model of VPR. To perform routing, it uses a global and detailed routing algorithm. The design of the fabric allows the algorithm to represent routing between CLBs as routing between switch matrices to which CLBs are connected. The global routing algorithm works like the VPR algorithm. Nets are initially routed with a greedy algorithm. Instead of un-routing all nets, only illegal nets are re-routed in an iteration. While using the same routing cost model as the VPR router, ROCR incorporates a small routing adjustment cost to all routing resources used by an illegal route. The routing adjustment cost discourages the greedy routing algorithm from selecting the same initial path in subsequent iterations. Once global routing is done, ROCR performs detailed routing which consists in assigning the channels (path in the switch matrix) used for each route. Two routes present a conflict when both routes pass through a given switch matrix and are assigned the same channel. This problem can be solved by a graph-coloring algorithm. The use of Brelaz's vertex coloring algorithm allows a linear time approximation which is good enough for solving the routing problem. ROCR takes advantage of the regular and basic structure of the switch matrix of a custom made fabric. It is an order of magnitude faster than VPR and uses an order of magnitude less memory.

To obtain a faster place and route algorithm, we explored several approaches. As JIT systems are typically used for high-level languages, we chose to restrict ourselves to fixed-width operators. We reduced the size of the problem by increasing the granularity of the operators, which operate on 16-bit integers. We also allowed algorithms to fail when problems are too complex, with a fallback invoking the more expensive algorithms.

Placement To simplify the placement algorithm, all operators are placed on a horizontal line whose height equals 4 CLB. Thus, the placement problem has only one dimension.

We implemented a classical simulated annealing algorithm. It takes about 120 ms for a circuit of 30 operators and 300 nets. This algorithm gives quite good results and routing always succeeded in our tests.

In our search for a faster algorithm, we attempted to implement a simpler algorithm which worked surprisingly well on our tests. We performed initial placement by flattening the expression tree in the topological order of the data dependencies. Then we performed a peep-hole optimization by trying every possible permutation of operators in a sliding window and kept the result with the lowest cost. This algorithm is greedy because it always keeps a better candidate when it finds one and never goes back to a worse solution. The algorithm loops until a stable state is reached. It usually needs about 3 iterations to find the best candidate, which takes less than 2 ms. The sliding window is illustrated in Figure 6. Notice that a CLB can have two overlapping operators (one for

each column of slices) and the routing matrix is shared by the two operators (Figure 5).

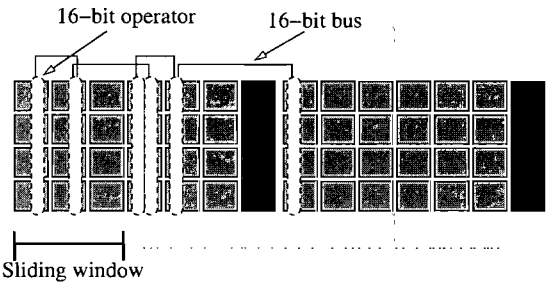


Fig. 6. Fast placement with peephole minimisation

This algorithm only requires a few kilobytes of memory. The results are not as good as the ones obtained by the simulated annealing algorithm but this does not affect the performance of the final circuit.

Routing Routing is more complex than placement. Figure 7 shows routing resources used in our algorithm. FPGA routing resources consist of switch matrices connected by external wires. There are various kinds of external wires: simple, double and hex. Simple wires connect direct neighbors, double wires connect the first and second neighbors, and hex wires connect third and sixth neighbors. Longer paths are routed through multiple switch matrices using multiple external wires.

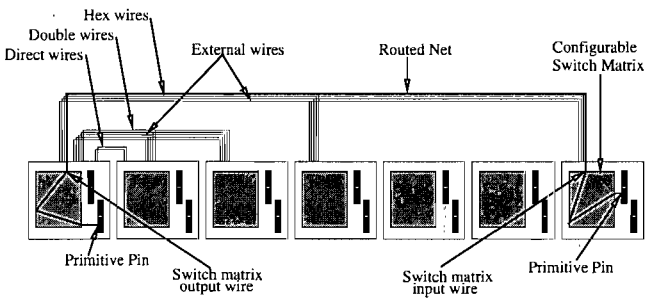


Fig. 7. Routing resources and a typical route

A typical route starts from a primitive pin, passes through a switch matrix, uses a wire to another switch matrix and ends at another primitive pin. All

programmable interconnect points (PIP) are in the switch matrix and nothing is configurable outside the CLB.

The routing algorithm is split into two levels: *external* and *internal* routing. The algorithm applies a sequence of greedy external routers that call different internal routers to manage the routing inside the switch matrices. By carefully pairing external and internal routers and by applying greedy and efficient algorithms first, it is possible to minimize the routing time and obtain an efficient routing.

Internal Router Paths inside a switch matrix and their corresponding Programmable Interconnects Points (PIP) are determined by the internal router. Internal routing is hard to achieve because the switch matrix is irregular, not fully connected and paths may be of various depths. This router finds the shortest path in the switch matrix from an input wire to an output wire. It takes into consideration all used resources and may fail if there is no path or if the routing is too complex.

The first internal router is implemented using a lookup table. The shortest paths from all inputs to all outputs of a switch matrix are pre-computed. Therefore, if a resource in the shortest path is already used, the routing simply fails.

The second internal router uses depth-first traversal. This algorithm is faster than Dijkstra's algorithm because paths are usually very short. The depth-first search was implemented by a set of mutually recursive functions that represent the connectivity of the switch matrix. Thus, there is no data manipulation, only fast code execution. We ordered the recursive calls to maximize the likelihood of reaching an output point.

External Router The external router drives the algorithm to determine where the next switch matrix is and which external wire is taken between switch matrices. As stated above, it uses an internal router as a helper to solve connections inside the switch matrix. External phases are sorted in such a way that most of the nets will be routed rapidly at the beginning while the remaining nets will be routed by a slower and more complex algorithm at the end.

The *direct* phase routes nets that can be routed only by using direct wires. Thus, only direct neighbors can be routed by this phase. The next phase, *general routing*, tries to route nets that only use one double or hex wire. As the placement minimizes the distance between neighbors, after these two phases, almost all nets are routed except those passing through more than one switch matrix or larger than 6 CLBs. So, the next phase is *greedy routing*. The idea is to find recursively the best path through switch matrices by jumping as close as possible to the target. The last phase uses *dynamic programming* to find a valid path and is able to route any net unless resource conflicts make it impossible.

Figure 8 shows examples of nets routed by each phase. As we see with the direct and general phases, only one external wire connecting two switch matrices is used. Greedy and dynamic routing use multiple external wires. We see the

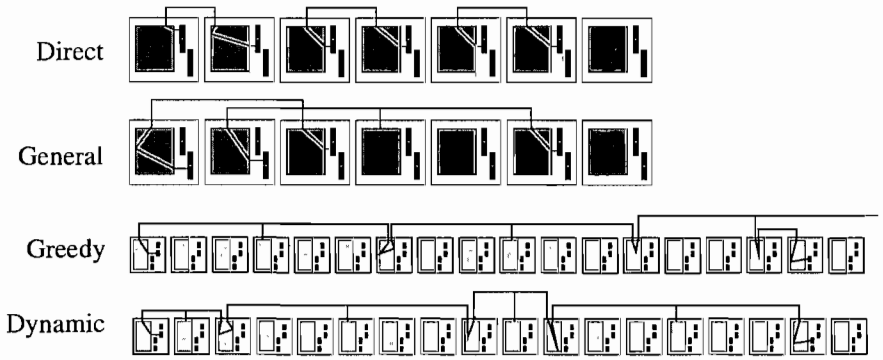


Fig. 8. Typical routes obtained with the 4 external routers

difference between greedy routing and dynamic programming routing: the greedy routing always jumps as close as possible. In the example, it jumps the longest possible path (6 CLBs) twice. The dynamic routing is able to find an alternative route but it is slower. It is used when conflicts occur with the greedy algorithm.

Nets cache Each external router keeps a cache of routed nets to determine if another net can be routed the same way by applying a translation. There are many cache hits because we use fixed-width operators and all the bits of a given operator are often routed in the same way.

5 Results

We synthesized the design on the ML310 demo board made by Xilinx. This system board has a Virtex-II Pro FPGA (2VP30) and 256 MB of external memory. The board has more components (e.g. PCI Bus, VGA, Ethernet connector...) but they were not used in our basic design; we only used the external memory, the FPGA and the serial link. The program is running on the processor and communicates through a console attached to the serial link.

Figure 9 shows a FPGA editor screenshot of a specialized intruction dynamically generated by our compiler. We observe the horizontal placement of 4 CLB high operators.

To achieve high processing rates it is important to have a close coupling of the processor and the dynamic instructions to minimize the communication costs. It is also important for the dynamic instructions to exploit parallelism, for example with a pipelined circuit. Because these aspects are orthogonal to the placement and routing problems tackled in this paper we have used a less efficient interface adequate for testing the correctness of the synthesized circuits. In our prototype the input and output of the dynamic instructions are not interfaced to the processor bus. The parameters and results are communicated through the

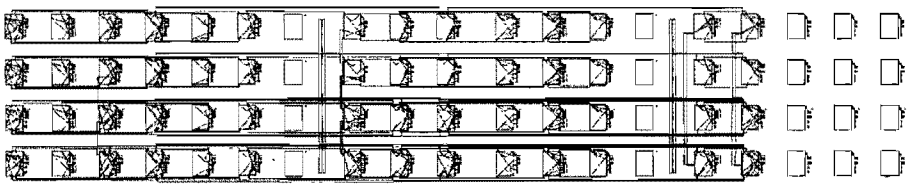


Fig. 9. Specialized instruction dynamically produced by the JIT compiler

Internal Configuration Access Port (ICAP). The parameters become constant values and the result is read back from the state of the output register. Given the relatively slow speed of the ICAP interface, application speed-ups are not possible. To solve this issue we must work-around Xilinx design flow limitations; this work is still in progress. Given our goal to show that compilation time can be low enough to support hardware JIT compilation, we focus our attention on the synthesis phases. Therefore, instead of benchmarking applications, we evaluate the time needed to perform the placement and routing which are the bottlenecks of synthesis.

slices	nets	Place (ms)		Route (ms)			
		S.A.	fast-place	dynamic	no-table	no-cache	fast-route
48	80	3	0	3	2	4	2
64	112	8	0	5	3	5	3
80	144	7	0	4	3	3	3
144	272	47	2	12	5	4	4
224	432	166	6	227	143	19	14
440	864	763	46	1161	385	101	50

Table 1. Place and route times for specialized intructions

Table 1 shows place and route times for expressions of various sizes (from 6 to 55 operators). For the placement, we compare the simulated annealing implementation with our fast implementation. We observe that our fast implementation is about 20 times faster. It is important to note that our simulated annealing implementation is already faster than current hardware synthesis tools because it also performs one-dimensional placement of fixed-width operators.

The last set of columns shows the routing time of expressions. By disabling the cache, we observe that the routing speed is halved. We also observe the speed-up obtained by using pre-computed tables for switch matrices by comparing the *fast-route* and the *no-table* columns. The *dynamic* column shows the worst case of our algorithm when all nets are routed with the dynamic programming router.

Typical routing tools use the Pathfinder and Dijkstra's algorithm. At each iteration, they re-route wires. Our algorithm is not iterative and routes each wire only once. Dijkstra's algorithm's speed should be similar to the dynamic

router speed. As VPR uses an iterative algorithm with Dijkstra's algorithm, its speed should be of the same order. Our algorithm does not iterate and only applies a sequence of greedy routers. Speed-ups come from the fact that most nets are routed by a faster router. Table 2 shows how many nets are routed by each internal and external router. Almost all nets are routed by the direct and general routers and a few nets really need a depth-first traversal to find a path. This explains why our algorithm is efficient.

nets	pre-computed tables				depth-first traversal			
	direct	general	greedy	dynamic	direct	general	greedy	dynamic
80	56	16	0	0	8	0	0	0
112	68	32	0	0	12	0	0	0
144	56	80	0	0	8	0	0	0
272	140	120	0	0	12	0	0	0
432	136	252	35	0	8	0	0	0
864	224	532	88	0	0	0	0	0

Table 2. Number of nets routed by each internal phase

6 Conclusion

We have described a hardware JIT compiler for dynamically reconfigurable FPGAs. At run time, the JIT compiles function closures to a hardware circuit which is downloaded to the reconfigurable zone of the FPGA. The JIT performs all the normal phases of hardware synthesis including placement and routing. To achieve good compilation speed we use a set of routing algorithms of increasing complexity which are tried in a cascade. Most if not all of the work is usually performed by the cheaper algorithms. The prototype we have built for the Xilinx Virtex II Pro FPGA demonstrates that modest sized functions can be compiled to hardware in a few milliseconds. Our experiments show that naive algorithms and lookup tables significantly speed-up placement and routing.

Various heuristics and naive fast algorithms may be adapted to FPGAs depending on their specific characteristics. We do not claim to have the best set of algorithms. Our claim is that compilation times compatible with JIT compilation are attainable on current reconfigurable FPGAs. We hope that our work will spur further research on hardware JIT compilation for general-purpose languages.

References

1. Etienne Bergeron, Marc Feeley, and Jean Pierre David. Toward On-Chip JIT Synthesis on Xilinx Virtex-II Pro FPGAs. In *50th International Midwest Symposium on Circuits and Systems/5th International Northeast Workshop on Circuits (MWCAS/NEWCAS), Montreal, Canada*, August 2007.

2. Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley, and Jean Pierre David. High-level synthesis for data-driven applications. In *IEEE International Workshop on Rapid System Prototyping*, pages 54–60, 2005.
3. Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.
4. G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, Washington, DC, USA, 1997.
5. Carl Ebeling, Larry McMurchie, Scott A. Hauck, and Steven Burns. Placement and routing tools for the Triptych FPGA. *IEEE Trans. Very Large Scale Integr. Syst.*, 3(4):473–482, 1995.
6. Marc Feeley. Gambit-C. http://www.iro.umontreal.ca/~gambit.
7. Marc Feeley and Guy Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.
8. Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.
9. Brian Greskamp and Ron Sass. A Virtual Machine for Merit-Based Runtime Reconfiguration. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.
10. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
11. R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM Transactions on Design Automation of Electronic Systems*, pages 659–681, July 2006.
12. R. Lysecky and F. Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Design Automation and Test in Europe Conference*, 2004.
13. R. Lysecky, F. Vahid, and S. X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. *Design Automation Conference*, pages 954–959, 2004.
14. R. Lysecky, F. Vahid, and S. X.-D. Tan. A study of the scalability of on-chip routing for just-in-time FPGA compilation. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 57–62, 2005.
15. Xiao ping Ling and Hideharu Amano. Performance evaluation of WASMII: a data driven computer on a virtual hardware. In *PARLE '93: Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, pages 610–621, London, UK, 1993. Springer-Verlag.
16. H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Reed Taylor. PipeRench: A virtualized programmable datapath. *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, pages 63–66, 2002.
17. G. Stitt and F. Vahid. Binary Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):34, 2007.
18. Xilinx. XAPP290: Two Flows for Partial Reconfigurable Core Based on Small Bit Manipulations. Technical report, Xilinx, September 2002.
19. Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Technical report, Xilinx, June 2005.

6.2 Compilateur Dynabit

Le compilateur Dynabit est responsable de la production à la volée de configurations matérielles pour l'architecture matérielle de Dynabit. Dans cette section, nous décrivons les phases importantes utilisées par le compilateur pour y parvenir. Par rapport à ce qui a été présenté à la conférence *Compiler Conference 2008*, de nombreuses modifications ont amélioré de manière significative les performances de la synthèse et nous croyons important d'expliquer nos nouvelles techniques.

6.2.1 Source

Le langage de programmation utilisé est le langage fonctionnel Scheme. L'utilisation du compilateur existant Gambit [Fee] et de son environnement d'exécution (*runtime*) nous a permis d'utiliser un ensemble de bibliothèques. Cependant, le langage contient des types (e.g. paire, liste, vecteur, ...) non supportés par le compilateur dynamique de Dynabit, qui ne reconnaît essentiellement que les calculs sur les entiers. Ainsi, les fonctions compilées dynamiquement sont restreintes à un sous-ensemble des fonctionnalités de Scheme. Le système pourrait être étendu pour supporter d'autres opérateurs, mais il s'avère suffisant pour faire des tests non-triviaux et valider notre approche.

Nous avons défini un ensemble d'opérateurs numériques qui effectuent des opérations élémentaires sur des entiers de 32 bits (e.g. `#%add`, `#%mul`, `#%shl`, `#%rol`, ...). Ainsi, les calculs voués à être transformés en matériel doivent s'exprimer en opérateurs 32 bits. Notons que cette limite est celle typiquement imposée par les architectures logicielles basées sur des processeurs 32 bits.

Pour simplifier la lecture des codes sources, nous avons supposé que les opérations arithmétiques et booléennes s'effectuent sur des *fixnums* de Gambit, et donc qu'elles sont représentables sur 32 bits, ce qui correspond à une déclaration "`(declare (fixnum))`" implicite. L'opérateur "+" correspond ainsi à l'opérateur de base "`#%add`" sans aucune modification de code.

Dynabit est intégré à l'environnement d'exécution Gambit, ce qui lui permet d'obtenir par introspection certaines informations contenues dans la représentation intermédiaire de celui-ci. L'utilisation de la primitive "`##decompile`" de Gambit permet de retrouver le code source

d'une fonction, puis l'environnement de la fonction est analysé afin d'y retrouver les paramètres candidats à l'évaluation partielle. La Figure 6.1 contient un exemple d'utilisation des primitives `##decompile` et `environment`.

```
1 (define make-adder
2   (lambda (a)
3     (lambda (b) (+ a b))))
4
5 (define inc (make-adder 1))
6
7 (pp (##decompile inc))    ;; --> (lambda (b) (+ a b))
8 (pp (environment inc))    ;; --> ((a . 1))
```

FIG. 6.1 – Introspection des fonctions avec Gambit

Nous croyons qu'il reste beaucoup de travail à faire pour obtenir un système capable d'exécuter l'ensemble du langage Scheme en matériel et nous espérons que des travaux futurs pallieront ces lacunes.

6.2.2 Évaluation partielle

L'évaluation partielle (EP) consiste en la simplification de certaines expressions d'un code, au moment de sa compilation, selon des paramètres connus au moment de la compilation. Dynabit fait l'évaluation partielle de la fermeture en cours de synthèse. Le premier bénéfice de cette passe est de fournir implicitement un système d'expansion au programmeur pour que les appels de fonctions soient évalués à la compilation (une approche similaire mais plus puissante que le *generate* de VHDL).

Ainsi, le code de la Figure 6.2, qui contient la boucle principale (`iter`) de l'algorithme de hash MD5 [Riv92] effectuant 64 étages de calcul, est déroulé en une expression simple ne contenant que les opérateurs de base. Cette expansion facilite l'écriture d'applications et s'apparente au concept de *generate* de VHDL.

```
1 (lambda (w1)  ;; MD5 partiel  (statique : w[0], w[2..15])
2   (define iter
3     (lambda (i a b c d w1)
4       (if (= i 64)
5           (and (= a (#%sub (u32vector-ref H 0) #x67452301))
6                (= b (#%sub (u32vector-ref H 1) #xEFCDAB89))
7                (= c (#%sub (u32vector-ref H 2) #x98BADCFE))
8                (= d (#%sub (u32vector-ref H 3) #x10325476)))
9           (let ((na (#%add b (#%rol (#%add a (F i b c d)
10                                            (K i) (W i w1))
11                                    (R i)))))
12             (iter (+ i 1) d na b c w1)))))
13   (iter 0 #x67452301 #xEFCDAB89 #x98BADCFE #x10325476 w1))
```

FIG. 6.2 – Code source de l'algorithme MD5 (partiel)

Le second bénéfice de l'EP est l'augmentation de la densité fonctionnelle de l'application. Certains paramètres peuvent être jugés temporairement constants si leur valeur est fixée pour une période de temps suffisamment longue. Le cas se produit souvent avec les boucles imbriquées dont les indices des boucles externes restent constants tout au long de l'exécution des boucles internes.

Afin d'illustrer ce bénéfice, prenons le cas de l'algorithme d'un briseur de mot de passe MD5 (*password cracker*), une application qui tire profit de l'accélération matérielle et de l'EP, dont la Figure 6.3 contient une comparaison des implantations avec et sans l'EP.

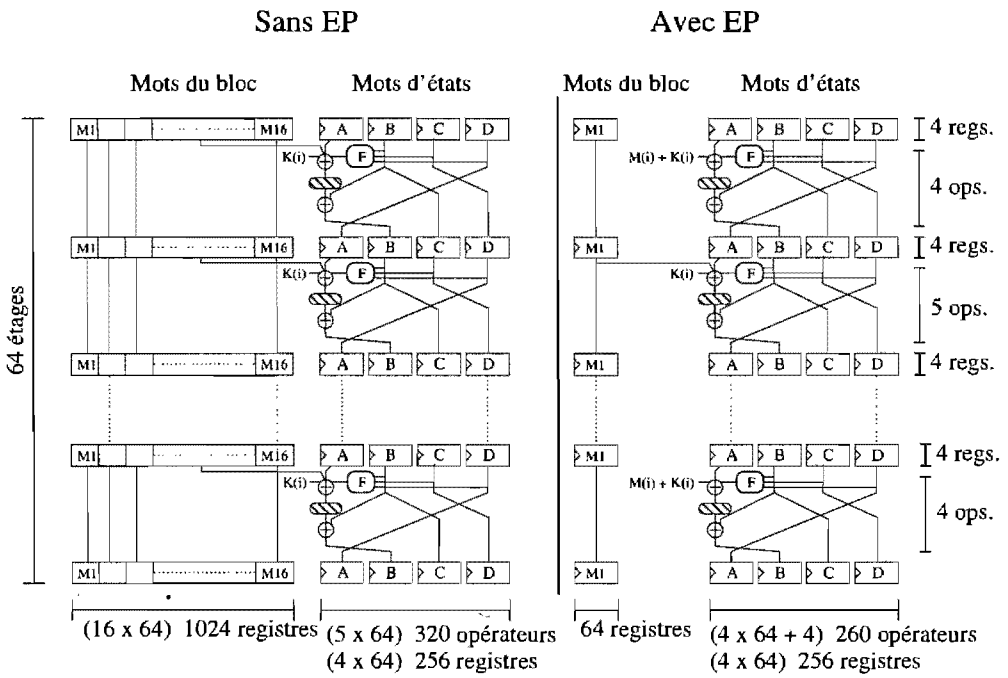


FIG. 6.3 – Évaluation partielle sur l'algorithme de hash MD5

Chaque étage de l'algorithme reçoit le bloc en cours de hashage, 16 mots de 32 bits (M1 à M16), ainsi que 4 mots d'état (A, B, C et D). Chaque étage doit utiliser des registres afin de minimiser la latence combinatoire d'un étage et ainsi atteindre la fréquence maximale. En pratique, il serait possible d'enlever des étages de registres mais au détriment de la fréquence d'horloge. Ainsi, simplement pour les registres du bloc, l'algorithme utilise 1024 registres de 32 bits, soit 32768 registres (*latches*), et à raison de deux registres par tranche et quatre tranches par CLB : 16384 tranches ou 4096 CLBs. L'inconvénient est que le VP30 de Xilinx, un modèle de taille moyenne, ne contient qu'une grille de 3680 CLBs (46 par 80) et ne peut donc pas contenir l'algorithme à sa fréquence maximale.

En regardant l'algorithme en cours d'exécution, nous observons que 15 des 16 registres restent constants pendant la période où l'autre registre parcourt l'espace total de sa clé (soit 32 bits) ; ceci correspond à environ 40 secondes à une fréquence de 100 MHz. Ainsi, l'idée de l'exécution avec EP consiste à produire une nouvelle configuration régulièrement (à toutes les 40 secondes) en supposant constants les paramètres dont la valeur ne change pas pendant le parcours de l'espace de la clé (e.g. 40 secondes). Ceci simplifie substantiellement la complexité du module produit. Comme le démontre la Figure 6.3, la simplification des registres permet d'obtenir un

module avec une densité fonctionnelle plus élevée qui tient dans l'espace disponible, et ce, à la fréquence maximale. Notons qu'une addition par étage est aussi évaluée par l'EP puisque les mots du bloc M(i) et les constantes de chaque étage K(i) sont connus statiquement. Ainsi, l'algorithme nécessite 64 registres de 32 bits et 260 opérateurs de 32 bits, soit 10368 opérations sur un bit : 5184 tranches ou 2592 CLBs. À noter que les 256 registres ne sont pas comptabilisés puisque chaque tranche contient une fonction combinatoire et un registre, ainsi les 260 opérateurs absorbent les 256 registres.

Il est théoriquement possible d'effectuer 100 millions d'essais par seconde à une fréquence de 100 MHz, ce qui est plus élevé que le 42 millions obtenus par mdcrack, le plus performant briseur MD5 logiciel, sur un processeur récent (2x XEON 3,2GHz). Pour permettre ces performances, le temps de synthèse doit être négligeable par rapport au temps d'utilisation d'une instance, 40 secondes. De plus, notons que la logique des nouveaux FPGAs peut avoir une fréquence de 400 MHz, ce qui permettrait potentiellement un facteur d'accélération de 10 par rapport à mdcrack.

Des accélérations de ce genre sont fréquentes dans le domaine des calculs numériques. Pour comprendre quelles applications tirent profit de ce mode d'exécution, il faut comprendre comment les opérateurs arithmétiques sont implantés en matériel.

La Figure 6.4 contient, à gauche, un multiplicateur matériel (4 bits) qui multiplie A par B. Son implantation consiste en une séquence d'additionneurs conditionnels qui accumulent des décalages de A selon la valeur des bits de B. La quantité de ressources nécessaires s'exprime par une équation quadratique en fonction de la taille des opérandes, ce qui devient coûteux en terme d'espace pour des opérandes de 32 bits.

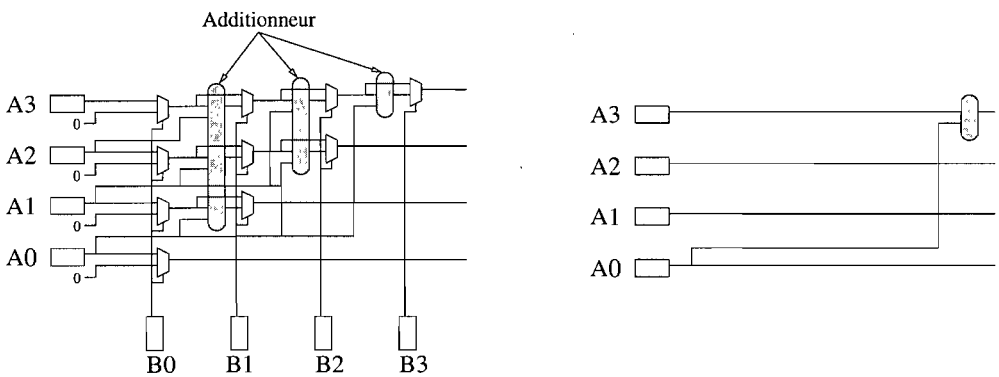


FIG. 6.4 – Évaluation partielle d'un multiplicateur 4 bits

Un multiplicateur par une constante peut s'évaluer partiellement et ainsi augmenter sa densité fonctionnelle, ainsi la Figure 6.4 contient, à droite, un exemple où l'opérande B est 9 (1001_2), et correspond au calcul : $A + 8A$. Notons que le nombre d'additionneurs éliminés équivaut au nombre de bits à zéro du paramètre B et que tous les multiplexeurs sont éliminés.

D'autres optimisations permettent de diminuer la taille d'un multiplicateur en utilisant l'algorithme de multiplication par une constante de Rice [BH94]. Par exemple, une multiplication par 15 comporte 4 additionneurs, puisque l'opérande B a 4 bits à 1. Toutefois, en utilisant l'équivalence $15x \equiv 16x - x$, le même calcul est implanté avec un seul additionneur et un décalage.

L'algorithme de Rice utilise la programmation dynamique pour déterminer une façon efficace d'implanter la multiplication. Il existe des techniques similaires, utilisées par Dynabit, pour implanter la division [AHS76] ainsi que le modulo par une constante.

6.2.3 Primitives matérielles

Les primitives matérielles sont les plus petites entités traitées par nos algorithmes de placement et de routage. Un ensemble limité de primitives matérielles nous a permis de supporter plusieurs opérateurs de base.

L'ensemble des primitives matérielles est :

ROL	Rotation constante
ADD	Addition
CADD	Addition conditionnelle
FLUT	Fonction Booléenne arbitraire
CMP	Égalité (e.g. égal, différent)
CMPL	Comparateur (e.g. plus petit, plus grand)

À noter que les opérateurs plus complexes peuvent se réduire en un ensemble de primitives et ainsi être traités par nos algorithmes. Par exemple, la multiplication se réduit à une séquence d'additionneurs conditionnels (CADD).

La Figure 6.5 contient trois de ces primitives. Nous y observons qu'une primitive mesure 8 CLBs de haut, qu'il est possible d'avoir deux primitives par CLB et que toutes les primitives sont alignées ; ce qui a comme conséquence d'aligner les routes entre les opérateurs.

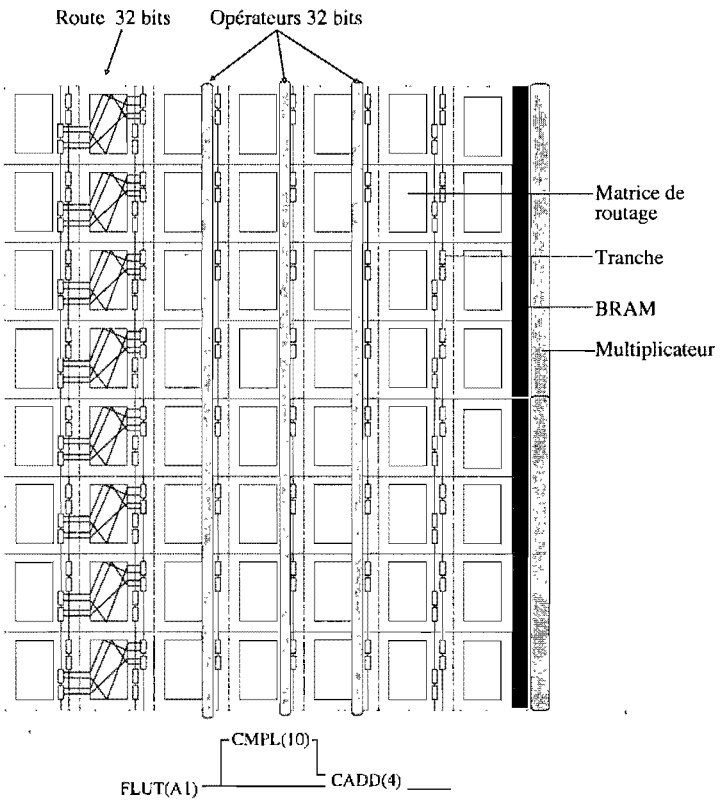


FIG. 6.5 – Primitives matérielles

Les primitives possèdent des paramètres configurables tels que l'équation des LUTs et l'entrée de la chaîne de retenue (BX). Cette flexibilité permet l'implantation de plusieurs opérations à partir d'une même primitive. De cette façon, tous les opérateurs booléens se réduisent à une primitive FLUT et la soustraction se réduit en l'addition du complément à deux. De plus, la configurabilité de ces primitives permet de les évaluer partiellement (*constant folding*) pour obtenir des primitives spécialisées (e.g. comparateur à une constante).

```
1 (and (= a (#%sub (u32vector-ref H 0) h0))
2      (= b (#%sub (u32vector-ref H 1) h1))
3      (= c (#%sub (u32vector-ref H 2) h2))
4      (= d (#%sub (u32vector-ref H 3) h3)))
```

FIG. 6.6 – Utilité de la spécialisation des primitives

La Figure 6.6 contient la branche terminale de l'algorithme MD5 qui effectue 4 comparaisons par les valeurs de H, constantes après EP, pour déterminer si l'entrée de la fonction donne la sortie (*hash*) recherchée. En pratique, ces opérations ne nécessitent qu'une primitive (CMP) à 4 entrées (a, b, c et d), dont les constantes ont été substituées dans l'équation des LUTs. Notons qu'un bon algorithme de correspondance technologique (*technology mapping*) est nécessaire pour tirer profit de cette flexibilité.

L'abstraction des primitives matérielles engendre une diminution de densité puisque certaines fusions de primitives ne sont plus possibles. Par exemple, avec les outils de Xilinx, il est possible d'utiliser séparément les LUTs et les registres d'une tranche, or dans notre cas, les registres reçoivent forcément les sorties des LUTs, vue l'implantation de nos primitives matérielles, et ne sont pas disponibles.

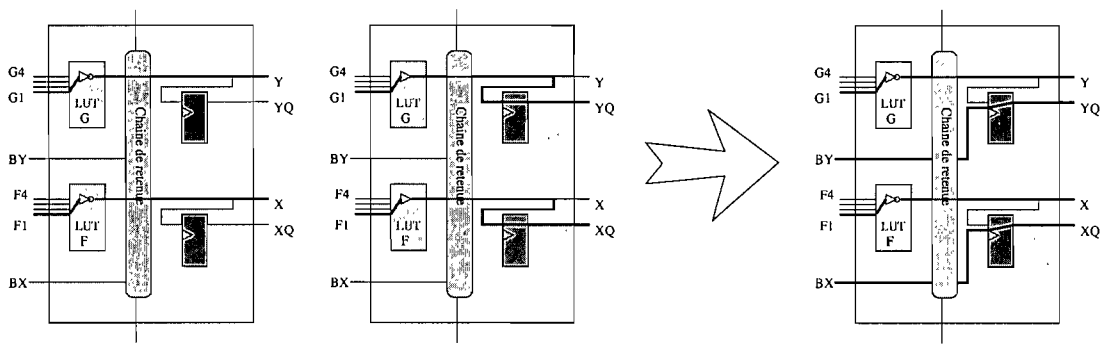


FIG. 6.7 – Superposition de primitives matérielles non supportée par Dynabit

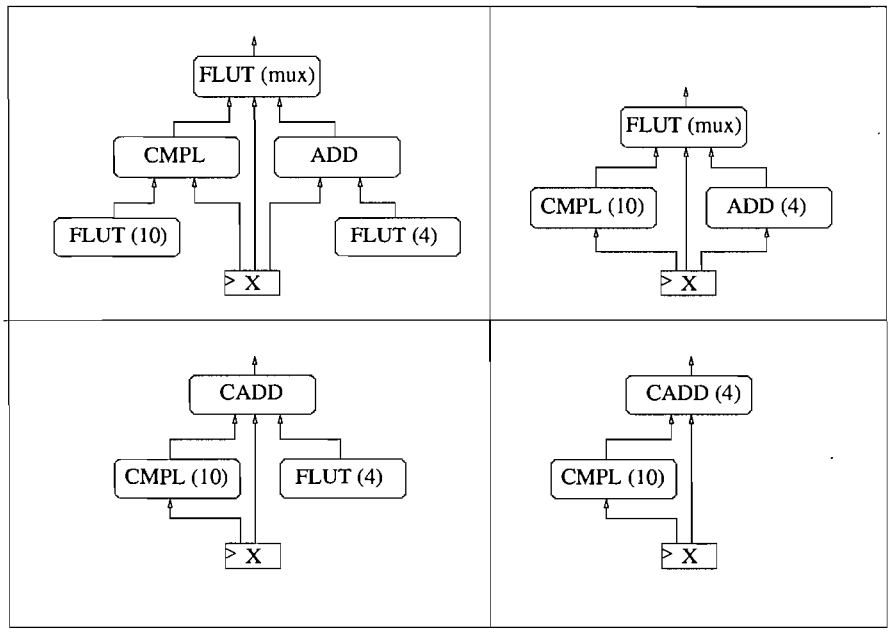
La Figure 6.7 montre un exemple où notre compilateur n'effectue pas de superposition : il est possible de superposer un registre et un inverseur dans une tranche. Nous croyons que cette perte de densité est compensée par le gain d'utilité d'une synthèse rapide. Dans la plupart des cas, les registres sont absorbés par l'opération qui précède le registre en utilisant la sortie enregistrée au lieu de la sortie combinatoire.

6.2.4 Correspondance technologique

La correspondance technologique (*technology mapping*) est la phase qui détermine avec quelles primitives physiques sont implantés les opérateurs de base. À titre d'exemple, un oulogique s'implante par un LUT.

Cette phase est similaire à la sélection d'instructions d'un compilateur qui consiste à déterminer quelles instructions utiliser pour une partie du calcul. Typiquement, elle est effectuée par un algorithme de programmation dynamique générant une couverture d'arbre, où l'arbre est la représentation intermédiaire du code.

Ces idées sont implantées par le compilateur GAMA [CCDW98] qui effectue la correspondance technologique et le placement en temps linéaire en utilisant un algorithme basé sur la programmation dynamique. Notre compilateur utilise des techniques inspirées des idées de ce projet.



(if (< x 10) (+ x 4) x)

FIG. 6.8 – Correspondance technologique d'une expression conditionnelle

La Figure 6.8 contient des exemples de correspondances entre les primitives matérielles et une expression conditionnelle. Il existe différentes implantations valides, et dans ce cas, la correspondance en bas à droite est la meilleure puisque l'expression conditionnelle, son test ainsi que l'addition n'utilise que deux primitives matérielles.

La qualité et la densité des correspondances produites par Dynabit reposent sur le nombre de patrons et leur qualité, et améliorer cet ensemble est une tâche qui est toujours d'actualité.

Il est à noter que la meilleure correspondance ainsi trouvée n'est pas forcément idéale pour la synthèse rapide puisqu'elle contient des primitives compactes, ce qui rend plus complexe le problème de routage par l'augmentation de conflits potentiels à l'entrée d'une tuile. Pour contrer ce problème, Dynabit effectue une passe de désoptimisation qui consiste à séparer ces primitives, lorsque l'espace et la latence le permettent.

6.2.5 Placement

Les algorithmes classiques de placement sont basés sur un recuit simulé [LMS06]. Les primitives sont positionnées aléatoirement, puis itérativement des permutations sont effectuées dans le but de minimiser une fonction de coût qui évalue la qualité du placement. Une température contrôle les déplacements qui n'améliorent pas le positionnement : les déplacements qui dégradent la qualité du placement sont de moins en moins fréquents au fur et à mesure que l'algorithme progresse, et tend ainsi vers une solution acceptable.

Notre algorithme de placement consiste à aplatir la représentation intermédiaire, sous forme d'arbre, en utilisant un tri topologique. La séquence de primitives ainsi obtenue est linéairement placée sur le FPGA.

La première version de Dynabit [BFD08] plaçait en une dimension, ce qui limitait la taille des instructions spécialisées. L'ajout du placement en deux dimensions permet des instructions larges au coût d'un placement plus complexe. Afin de minimiser la dégradation des performances de l'algorithme, nous avons conservé un placement linéaire en supposant que les rangées subséquentes sont en fait une suite logique de la première rangée. Ainsi, l'algorithme de placement effectue son travail en supposant une rangée continue mais l'espace utilisé physiquement est en deux dimensions.

Les mauvais placements dégradent considérablement la performance de l'algorithme de routage. Par conséquent, une passe applique des correctifs afin d'éviter les cas problématiques.

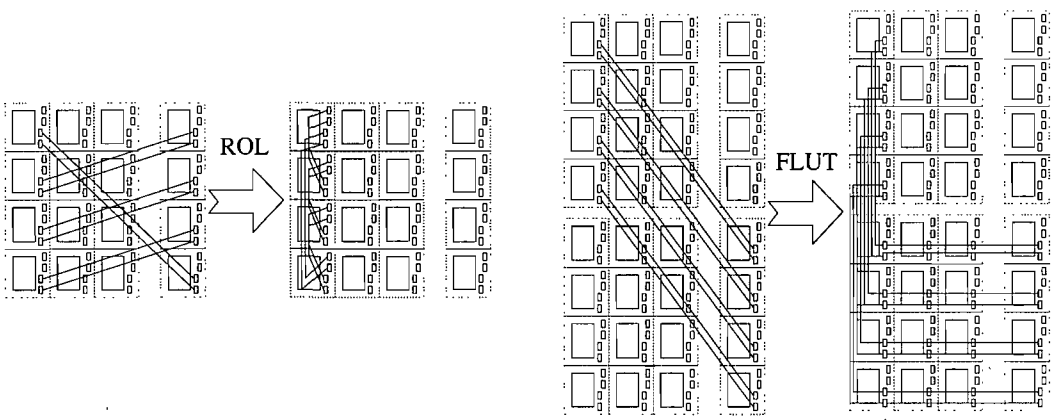


FIG. 6.9 – Correctifs pour faciliter le routage

Afin de simplifier le routage, les primitives doivent idéalement être alignées, mais certains mauvais placements les désalignent : la Figure 6.9 contient deux exemples fréquents ainsi que leur correctif. Parce qu'elles effectuent des opérations au niveau des bits, les rotations produisent des routes désalignées qui sont corrigées en imposant le positionnement de la source et de la destination à la même colonne. Dans le cas problématique où la primitive possède plusieurs destinations, une primitive identité est ajoutée à la sortie de la rotation. L'ajout d'une primitive identité est aussi utilisé entre les primitives non positionnées sur la même rangée.

L'application de ces correctifs diminue la complexité du routage car ces cas spéciaux sont prévus par l'algorithme de routage pour le routage vertical dans une même colonne. La perte d'espace nous paraît un compromis acceptable pour le gain de vitesse de synthèse ainsi obtenu.

6.2.6 Routage

Les algorithmes classiques de routage sont itératifs et basés sur la négociation de congestion (e.g. PathFinder [EMHB95, ME95]). Essentiellement, toutes les routes sont résolues, en ignorant les conflits, par l'algorithme de Dijkstra qui trouve le chemin le moins coûteux. Par la suite, le coût des ressources en conflits (e.g. utilisées par plusieurs routes) est augmenté pour inciter les routes à emprunter un autre chemin. L'algorithme itère sur ces étapes jusqu'à ce qu'aucun conflit ne soit présent. Le temps d'exécution de cette phase domine largement celui des autres phases. Par conséquent, une approche plus agressive est utilisée par Dynabit afin de réduire le temps de routage.

L'approche utilisée par Dynabit [BFD08] consiste à appliquer une cascade d'algorithmes ordonnés des plus rapides aux plus complexes dans le but de trouver rapidement un chemin pour la majorité des routes. Nous avons séparé le routage interne et le routage externe, puis optimisé différents algorithmes pour chacun. Les résultats ainsi obtenus étaient plus performants que les algorithmes classiques et, en conséquence, applicables à la synthèse dynamique.

Dans notre quête de performance, nous avons étudié les détails de la matrice de routage, malheureusement non documentée. Nous voulons comprendre les performances obtenues par nos algorithmes et tenter, si possible, de les améliorer. De nombreuses observations ont permis d'apporter des modifications à nos techniques et ainsi de grandement améliorer nos performances.

Dans cette section, nous détaillons nos observations et expliquons comment le routage peut

s'effectuer efficacement sur un VirtexII-Pro.

Ressources de routage

L'énumération de l'ensemble des ressources de routage se trouve dans le rapport produit par l'outil XDL. Afin d'y comprendre la connectivité, il est possible de faire un parallèle entre le rapport et les ressources avec l'outil FPGA_Editor de Xilinx.

Le FPGA contient un ensemble de fils statiques, regroupés en bus, et disposés de manière régulière sur toute la surface. Ces fils sont statiques et, par conséquent, ne peuvent être modifiés. La Figure 6.10 contient un exemple des fils statiques présents dans le VirtexII-Pro. Nous y observons des fils dédiés qui relient un CLB aux tuiles voisines. D'autres bus, de largeur dix, orientés dans les quatre directions : nord, sud, est et ouest, ont une longueur de deux pour les fils "doubles" et de six pour les fils "hex".

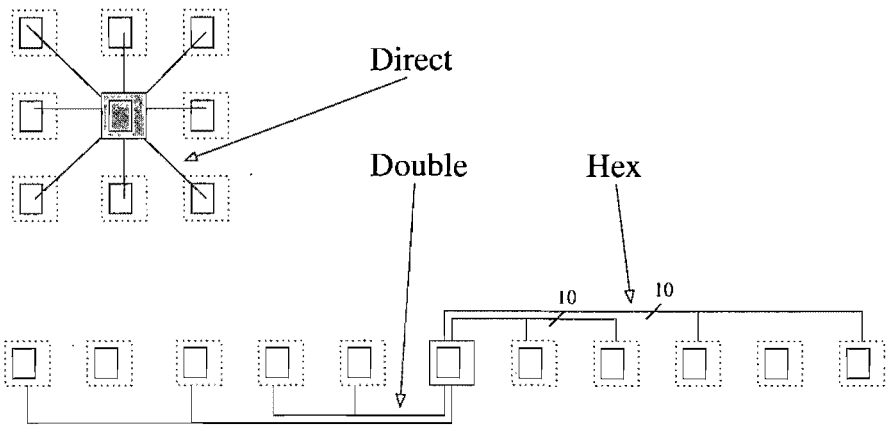


FIG. 6.10 – Fils statiques du VirtexII-Pro (direct, double et hex)

Les matrices de routage contiennent un ensemble de points programmables, appelés PIPs (*Programmable Interconnect Point*), qui activent le passage d'un signal entre deux fils statiques. Une route est construite par l'activation d'une série de PIPs afin de former un chemin continu d'une primitive à une autre. Il est à noter que les points programmables sont nécessairement à l'intérieur d'une matrice de routage, et ainsi ne peuvent être à l'extérieur d'une tuile.

La route ci-dessous est un exemple contenant des PIPs, représentés par la flèche simple ($\rightarrow$) et des fils statiques (parfois appelés connexions), représentés par la flèche double ($\Rightarrow$).

```
R1C1 X0 → R1C1 E2BEG0 ⇒ R1C3 E2END0 → R1C3 F1_B0
```

Comme les tuiles peuvent être déduites par rapport à l'origine de la route et aux fils empruntés, nous évitons de les écrire pour alléger la syntaxe et la compréhension. Ainsi, la route suivante est équivalente à la précédente si et seulement si son origine est la tuile `R1C1`.

```
X0 → E2BEG0 ⇒ E2END0 → F1_B0
```

Nous avons choisi un sous-ensemble des ressources de routage afin de tirer profit de la régularité du FPGA. En évitant les cas irréguliers, il nous est possible d'effectuer des translations de route, et ainsi obtenir des algorithmes de routage relatifs. La Figure 6.11 contient les ressources de routage ainsi que certains fils statiques que nous utilisons. Nous y observons une tuile (CLB), sa matrice de routage ainsi que ses quatre tranches.

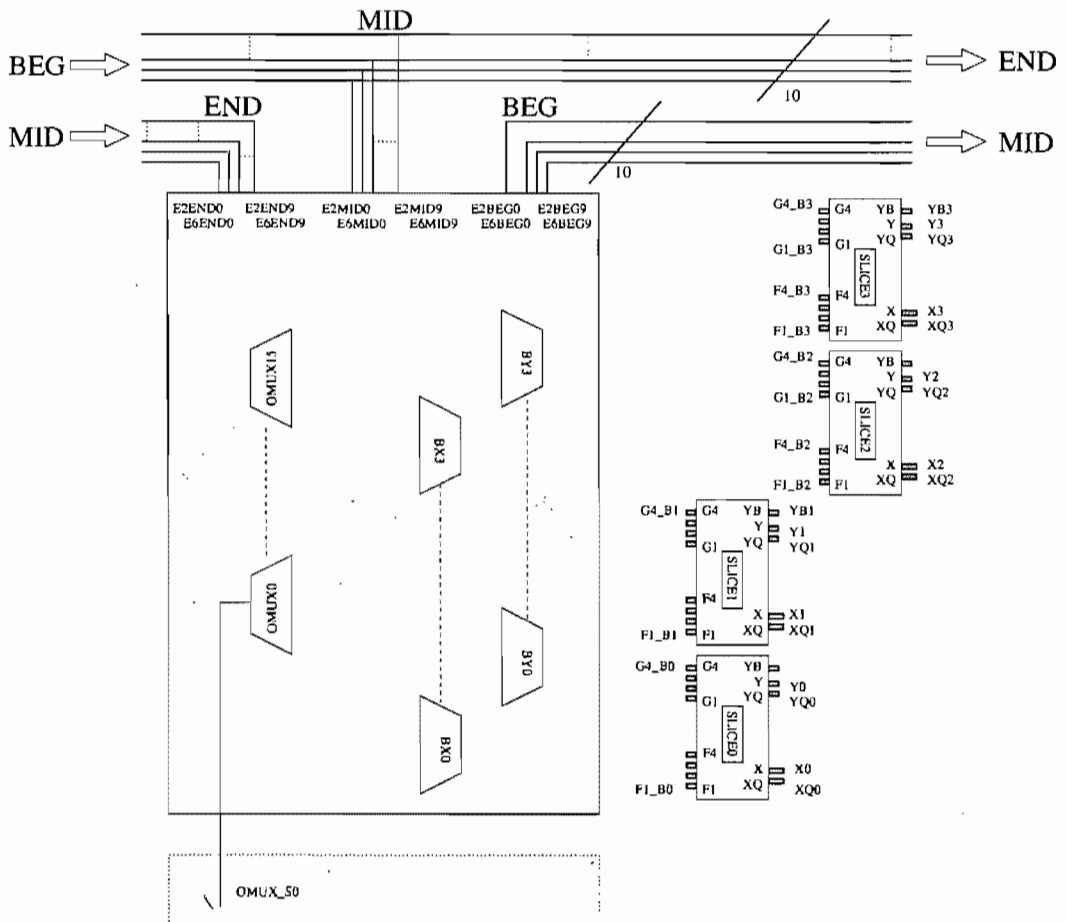


FIG. 6.11 – Ressources de routage des CLBs du VirtexII-Pro

Les tranches peuvent implanter une fonction combinatoire sur deux bits, deux équations de 4 bits vers 1 bit. Elles possèdent deux bus d'entrée de quatre bits (F1...F4 et G1...G4), ainsi que les sorties combinatoires (X et Y), les sorties enregistrées (XQ et YQ) et les sorties de la chaîne de retenue (XB et YB). Les entrées et les sorties des tranches sont connectées à des fils statiques nommés selon la tranche : l'entrée F1 de la tranche SLICE0 est connectée au fil F1_B0 tandis que celle de la SLICE1 est connecté au fil F1_B1.

À la Figure 6.11, nous pouvons aussi observer les bus "doubles" et "hex" qui entrent et sortent de la matrice de routage. La nomenclature des fils permet de déduire leur orientation, leur longueur ainsi que leur source et leur destination. Ainsi, le fil S2BEG4 est un fil de longueur deux (e.g. double) qui se dirige vers le sud, et le fil E6BEG0 est un fil de longueur six (e.g. hex) qui se dirige vers l'est.

En groupant certains PIPs par leur destination, nous avons déduit que la matrice de routage possède des multiplexeurs d'entrées et de sorties : 16 multiplexeurs de sorties (OMUX0...OMUX15) et 8 multiplexeurs d'entrées (BX0...BX3 et BY0...BY3). À noter que les routes ne sont pas obligées de les utiliser ; ces multiplexeurs permettent une plus grande connectivité. Voici les connexions possibles au travers des multiplexeurs OMUX2 et BX0 :

XQ0 XB0 X0		W2BEG1
YQ0 YB0 Y0		W6BEG1
XQ1 XB1 X1		S2BEG0 S2BEG1
YQ1 YB1 Y1	→ OMUX2 →	S6BEG0 S6BEG1
XQ2 XB2 X2		F4_B0 G4_B0
XQ2 XB2 X2		G1_B1 F1_B1
XQ3 XB3 X3		F4_B2 G4_B2
XQ3 XB3 X3		F1_B3 G1_B3
E2BEG1		
E2MID0 E2MID1 E2MID2		
E2END0 E2END1 E2END2		
W2BEG0		
W2MID0 W2MID1		F3_B0 G3_B0
W2END0 W2END1		F2_B1 G2_B1
N2BEG1	→ BX0 →	F3_B2 G3_B2
N2MID0 N2MID1		F2_B3 G2_B3
N2END0 N2END1		
S2BEG0 S2BEG2		
S2MID0 S2MID1 S2MID2		
S2END0 S2END1 S2END2		
OMUX_W1 OMUX_E2		
OMUX_EN8 OMUX_N10 OMUX_NW10		

D'autres ressources existent mais leur compréhension n'est pas requise pour l'analyse de nos heuristiques : les fils dédiés pour l'horloge, où encore les longs fils qui traversent complétement en hauteur et en largeur le FPGA.

Grammaire

Le routage est généralement vu comme un problème de graphe sur lequel on tente de trouver le chemin le plus court. Nous utilisons un autre formalisme pour définir le problème : une grammaire qui exprime la connectivité du graphe sous forme de règles de grammaire. De cette façon, trouver un chemin consiste à déterminer un arbre de dérivation.

Notons que la grammaire donnée est un sur-ensemble des chemins valides. Par conséquent, lors de la dérivation, il faut s'assurer de l'existence en matériel des PIPs ainsi que de leur disponibilité. Il en découle que toutes les routes produites par notre algorithme sont dérivables de notre grammaire, mais cela n'exclut pas qu'il soit possible d'en construire d'autres en utilisant les ressources irrégulières.

Nous pouvons dire que la taille de la grammaire est proportionnelle à la quantité de code nécessaire pour trouver une dérivation. Ainsi, il est possible d'avoir une grammaire avec une règle de tous les fils vers tous les fils ou encore d'avoir une grammaire avec une règle pour chaque connexion. Le déroulement de la grammaire permet de mettre en évidence des propriétés et ainsi permettre l'abstraction d'heurisiques. Donc, un compromis entre la taille et l'efficacité doit être déterminé afin d'obtenir un bon algorithme de routage.

Les premiers éléments de la grammaire sont les cas terminaux : les entrées (1), les sorties (2). les fils doubles et hex (3) et leurs connexions (4) ainsi que les fils dédiés (5).

FG	::=	`Fj_Bi` $(i \in 0\ldots3,\ j \in 1\ldots4)$	(1)
XY	::=	`Xi` | `Yi` $(i \in 0\ldots3)$	(2)
XYQ	::=	`XQi` | `YQi` $(i \in 0\ldots3)$	
YB	::=	`YBi` $(i \in 0\ldots3)$	
BXY	::=	`BXi` | `BYi` $(i \in 0\ldots3)$	
BEG	::=	`N2BEGi` | `S2BEGi` | `E2BEGi` | `W2BEGi` $(i \in 0\ldots9)$	(3)
	::=	`N6BEGi` | `S6BEGi` | `E6BEGi` | `W6BEGi` $(i \in 0\ldots9)$	
MID	::=	`N2MIDi` | `S2MIDi` | `E2MIDi` | `W2MIDi` $(i \in 0\ldots9)$	
	::=	`N6MIDi` | `S6MIDi` | `E6MIDi` | `W6MIDi` $(i \in 0\ldots9)$	
END	::=	`N2ENDi` | `S2ENDi` | `E2ENDi` | `W2ENDi` $(i \in 0\ldots9)$	
	::=	`N6ENDi` | `S6ENDi` | `E6ENDi` | `W6ENDi` $(i \in 0\ldots9)$	
Conns	::=	$BEG \Rightarrow (MID \mid END)$	(4)
OMUX	::=	`OMUXi` $(i \in 0\ldots9)$	(5)
*OMUX_*	::=	`OMUX_i` $(i \in 0\ldots9)$	
ConnsD	::=	`OMUXi` $\Rightarrow$ `OMUX_i` $(i \in 0\ldots9)$	

Une route peut être *locale* (6) si la source et la destination se trouvent à l'intérieur de la même tuile, *directe* (7) si la destination se trouve à l'intérieur d'une tuile voisine, ou *générale* (8) dans les autres cas.

Route	::=	*Locale*	(6)
	::=	*Directe*	(7)
	::=	*Tete Corps Queue*	(8)

Tel que la grammaire le démontre, une route locale (9) peut être un PIP direct, un PIP suivi d'un bond par un fil, ou encore un PIP suivi d'un bond et d'un multiplexeur d'entrée. Notons que la longueur de la route a une profondeur maximale de trois PIPs.

Local	::=	$(XY \mid XYQ \mid YB) \rightarrow [\ BEG \rightarrow]\ [\ BXY \rightarrow]\ FG$	(9)
Direct	::=	$(XY \mid XYQ \mid YB) \rightarrow OMUX$	(10)
		$\Rightarrow OMUX\_ \rightarrow [\ BEG \rightarrow]\ [\ BXY \rightarrow]\ FG$	

Exemples :

`X0` → `F4_B1`
`X0` → `N2BEG9` → `F1_B0`
`X0` → `N2BEG9` → `BX1` → `F2_B1`

Les routes directes (10) passent par un multiplexeur de sortie pour se connecter à la tuile

voisine, et utilisent les mêmes règles que le routage local pour compléter le chemin.

Exemples :

X0 → OMUX2 ⇒ OMUX_E2 → F1_B1
X0 → OMUX0 ⇒ OMUX_S0 → BX0 → F2_B1

Les routes générales se décomposent en trois segments : la tête, le corps et la queue. La tête de la route est un chemin qui relie la source à un fil double ou hex, et de la même manière, la queue relie un fil double ou hex à la destination. La règle corps consiste à emprunter des connexions puis faire des bonds via différentes matrices de routage afin de relier la tête à la queue.

Tete	::=	$XY \rightarrow [\ OMUX \mid (OMUX \Rightarrow OMUX\_) \rightarrow]\ BEG$
	::=	$XYQ \rightarrow OMUX\ [\Rightarrow OMUX\_\ ] \rightarrow BEG$
	::=	$YB \rightarrow OMUX\ [\Rightarrow OMUX\_\ ] \rightarrow BEG$
Queue	::=	$END \rightarrow [\ BEG \rightarrow]\ [\ BXY \rightarrow]\ FG$
Bond	::=	$(MID \mid END) \rightarrow BEG$
Corps	::=	$Conns\ (\ Bond\ Conns\ )^*$

Exemples :

X3 → E6BEG0 ⇒ E6END0 → E2BEG0 ⇒ E2MID0 → BX2 → F3_B1
X3 → OMUX2 → S6BEG0 ⇒ S6MID0 → E2BEG0 ⇒ E2MID0 → BX2 → F3_B1

Nous pouvons faire un parallèle entre ces règles et les concepts de routage interne et externe que nous avons présentés [BFD08] : le routage interne correspond aux règles *Tete, Queue* et *Bond* tandis que le routage externe correspond à la règle *Corps.*

Observations

Dans cette section, nous résumons certaines observations utilisées par nos heuristiques, plus particulièrement celles spécifiques aux règles de routage de la tête et de la queue, et puis des bonds puisque les optimisations réalisées grâces à celles-ci sont les plus fréquentes.

Nous définissons l'indice d'un fil comme son indice dans le bus : E2BEG3, E6BEG5 et W2BEG7 ont respectivement les indices 3, 5 et 7.

Notons que les connexions suivantes, introduites dans le but d'avoir une grammaire compacte, sont impossibles car elles doivent avoir le même indice et faire partie du même bus :

E2BEG0 ⇒ E6END0
E2BEG0 ⇒ E2END3
W2BEG0 ⇒ E2END0

Nous étions porté à croire que la matrice de routage avait un réseau de connexions fortement connecté, mais nos observations nous ont démontré que la connectivité y était limitée mais structurée.

Notons que les sorties combinatoires (X et Y) peuvent avoir un chemin direct, ce qui n'est pas le cas pour les sorties enregistrées et les sorties des chaînes de retenue qui doivent absolument passer par un multiplexeur de sortie. Ainsi, les primitives enregistrées ont une plus grande probabilité de conflit de ressources sur les multiplexeurs de sortie et, si possible, ne doivent pas être positionnées dans une même tuile.

Notons aussi que les multiplexeurs de sorties peuvent acheminer n'importe quelle sortie des tranches et sont utilisés pour accroître la connectivité.

Le Tableau 6.1 résume la connectivité des sorties combinatoires.

Fils	Indice Est/Ouest	Indice Nord/Sud
Y3	5, 8	6, 8
X3	0, 2	0, 3
Y2	4, 7	5, 7
X2	1, 3	1, 4
Y1	1, 4	2, 4
X1	5, 7	5, 8
Y0	0, 3	1, 3
X0	6, 8	6, 9

TAB. 6.1 – Définition des fonctions EO et NS

Ainsi, il est possible de se connecter sans utiliser de multiplexeur de sortie le fil X0 aux fils E2BEG6, E6BEG6, E2BEG8 et E6BEG8 pour effectuer un routage direct dirigé vers l'est. Nous représentons la connectivité par les fonctions $EO(fil) \rightarrow \{indice\}$ et $NS(fil) \rightarrow \{indice\}$, ainsi $EO(Y3) \rightarrow \{5,8\}$ et $NS(X0) \rightarrow \{6,9\}$.

Nous observons au Tableau 6.2 une connectivité régulière et limitée pour les entrées des tranches.

Fils	Indice	Fils	Indice
SLICE 1		SLICE 3	
F4_B1 G4_B1	7, 8, 9	F4_B3 G4_B3	7, 8, 9
F3_B1 G3_B1	2, 3, 4	F3_B3 G3_B3	2, 3, 4
F2_B1 G2_B1	5, 6, 7	F2_B3 G2_B3	5, 6, 7
F1_B1 G1_B1	0, 1, 2	F1_B3 G1_B3	0, 1, 2
SLICE 0		SLICE 2	
F4_B0 G4_B0	0, 1, 2	F4_B2 G4_B2	0, 1, 2
F3_B0 G3_B0	5, 6, 7	F3_B2 G3_B2	5, 6, 7
F2_B0 G2_B0	2, 3, 4	F2_B2 G2_B2	2, 3, 4
F1_B0 G1_B0	7, 8, 9	F1_B2 G1_B2	7, 8, 9

TAB. 6.2 – Définition de la fonction IN

Nous définissons la fonction IN comme $IN(fil) \rightarrow \{indice\}$, et par conséquent $IN(F1\_B0) \rightarrow \{7, 8, 9\}$. Notons que les indices des tranches de la colonne de gauche sont identiques à ceux de la colonne de droite. De plus, les indices sont inversés pour les tranches du haut et du bas.

Lorsque l'indice du fil de destination n'est pas approprié, l'utilisation d'un bond ou d'un mutiplexeur d'entrée est requise pour ré-orienter la route.

En ce qui a trait aux matrices de routage, leur connectivité est aussi limitée pour les bonds. Les bonds possibles pour une route horizontale orientée vers l'est sont :

$$\begin{array}{ll} \texttt{E2END}_n \rightarrow \texttt{E2BEG}_n & \texttt{E6END}_n \rightarrow \texttt{E2BEG}_n \\ \texttt{E2END}_n \rightarrow \texttt{E2BEG}_{(n-2)} & \texttt{E6END}_n \rightarrow \texttt{E2BEG}_{(n-2)} \\ \texttt{E6END}_n \rightarrow \texttt{W2BEG}_{(n)} & \end{array}$$

Nous observons qu'il est possible de faire un chemin continu avec le même indice, ou un bond avec un décalage d'indice. Il faut noter que le décalage n'est pas le même dans toutes les directions. De plus, nous observons qu'une route qui passe par un fil double ne peut retourner sur un fil hex.

De nombreuses autres observations ont été faites afin d'ajuster nos heuristiques, mais celles démontrées ci-haut sont suffisantes pour comprendre nos améliorations.

Heuristiques

La structure de l'algorithme de routage est constituée d'un ensemble d'algorithmes de routage spécialisés pour différentes règles de la grammaire. Par exemple, la règle *Tete* peut se résoudre

par une recherche en profondeur, bornée à une profondeur de 3. Une passe est obtenue en combinant ces algorithmes. L'objectif de nos heuristiques est de tenter de couper des branches de recherche et converger rapidement vers une solution.

Les opérateurs sont alignés, par conséquence, la majorité des routes sont horizontales. Ainsi, dans cette section, nous nous intéressons particulièrement aux routes générales et horizontales.

Le routage de la tête ou de la queue est terminal dans le sens où il n'est pas nécessaire de dériver une autre règle. Comme l'énumération de ces routes est possible, nous les avons toutes dérivées à partir de la grammaire et nous avons produit un arbre de décision efficace, dirigé par des tables et par du code déroulé. Puisque ce code est exécuté souvent, sa performance a été optimisée en minimisant le nombre d'accès mémoire et l'allocation mémoire.

La complexité algorithmique du routage provient de la construction du corps de la route puisque le nombre de chemins possibles est élevé. Pour simplifier sa construction, nous avons ignoré les bonds avec un décalage d'indice. Cette simplification nous permet d'obtenir une régularité pour tous les bonds d'une route : ils auront tous le même indice. En connaissant l'indice d'entrée et de sortie du corps d'une route, il est possible de déterminer d'avance si le routage d'une extrémité est impossible et ainsi éviter de construire un corps complexe pour finalement obtenir un échec. De plus, l'élimination des bonds avec décalage nous a permis d'énumérer les routes horizontales de moins de 15 tuiles et d'en produire un arbre de décision efficace.

À partir de nos observations, il est possible de guider en premier le routage vers les chemins possédant une bonne probabilité de réussite et un coût moindre. Ainsi, une route ayant comme source X0 et comme destination F1_B0 devrait utiliser l'indice 8 ($EW(X0) \cap IN(F1\_B1) \rightarrow \{8\}$) pour éviter l'utilisation de ressources supplémentaires. De la même manière, une route entre X0 et F2_B0 implique forcément l'utilisation d'un multiplexeur puisqu'ils ne possèdent pas d'indice commun.

Passes

L'algorithme de routage applique une série de passes de plus en plus complexes avec lesquelles nous essayons de trouver un chemin aux routes. L'utilisation de plusieurs passes a pour objectif d'encourager le choix des routes qui utilisent le moins de ressources.

La première passe trouve un chemin aux routes qui ne sont pas parfaitement horizontales car

ce sont les plus difficiles à résoudre et évite ainsi qu'une autre passe utilise une des ressources requises. Ce cas se produit pour les rotations et les renvois à la ligne suivante : la liste exhaustive de ces chemins est courte et les patrons sont tous inclus dans le compilateur.

La seconde passe utilise des heuristiques que nous avons prévues pour certains cas fréquents. À ce stade, nous empêchons l'utilisation des multiplexeurs d'entrées ou de sorties afin d'emprunter, si possible, une alternative moins coûteuse. La troisième passe permet l'utilisation des multiplexeurs d'entrées et de sorties et, par conséquent, résout la plupart des conflits de ressources rencontrés par la précédente passe par l'utilisation de route alternative (différents indices).

À ce stade, généralement, toutes les routes ont un chemin établi et l'algorithme du plus court chemin de Dijkstra est utilisé pour déterminer une route à ce qui reste. Il est à noter qu'une route qui a trouvé un chemin n'est jamais modifiée, et que l'algorithme n'a aucune itération. C'est grâce à ces propriétés que notre algorithme atteint de bonnes performances.

Résumé

Il devient évident que le gain de performance obtenu est au sacrifice de la portabilité de notre technique qui est spécialisée pour les VirtexII-Pro. Des heuristiques spécialisées et basées sur des observations manuelles ont permis d'améliorer substanciellement les performances de l'algorithme de routage. Cependant, les architectures des nouvelles familles (Virtex 4 et 5) ont beaucoup en commun avec les VirtexII-Pro et nous n'entrevoyons pas de problèmes particuliers à adapter nos heuristiques à ces nouvelles familles.

6.3 Conclusion

L'évolution de nos techniques de synthèse rapide nous ont finalement conduits à des performances qui satisfont les besoins pour la synthèse à la volée.

L'évaluation partielle de modules matériels peut maintenant s'appliquer à une plus large classe d'applications et ainsi augmenter leurs performances. Certaines applications, trop larges pour s'implanter sur un FPGA, peuvent utiliser un modèle d'exécution dynamique et diminuer considérablement leurs tailles afin de leur permettre de répondre aux contraintes d'espace et de latence.

L'utilisation d'algorithmes de placement et de routage agressifs, conçus dans l'optique de sacrifier la densité pour un gain en vitesse, est un aspect essentiel pour obtenir des temps de synthèse adéquats. La complexité algorithmique et le temps d'exécution élevé de l'algorithme de routage implique que celui-ci soit fortement adapté à l'architecture matérielle.

Chapitre 7

Résultats

Ce chapitre vise à analyser les qualités de notre approche de façon empirique. Dans un premier temps nous comparons notre approche avec les outils de synthèse existants, spécifiquement ceux de Xilinx qui sont représentatifs des outils disponibles commercialement. Nous nous intéressons à deux propriétés de la synthèse : le temps de synthèse et la densité. Par la suite, les différentes phases de la synthèse sont séparément analysées afin d'identifier lesquelles sont coûteuses. Nous évaluons aussi les coûts de communication entre le processeur et les instructions spécialisées afin d'expliquer les différentes méthodes de communication utilisées pour absorber la latence. Finalement, des cas d'utilisation de la synthèse à la volée sont présentés afin de démontrer l'intérêt de l'évaluation partielle qui justifie la synthèse à la volée.

Les temps d'exécution ont été mesurés sur un MacBook Pro (Intel Core Duo, 2 GHz, 2 GB RAM). Dans certains cas, les temps d'exécution sur le FPGA (XC2VP30-ff896-5, PPC405, 350 MHz, 256 Mo RAM) sont présentés afin de comparer ces performances avec un processeur récent. Nous jugeons que le PPC405 sur le FPGA est dépassé (sa conception date d'il y a au moins 10 ans) et que dans le futur les FPGAs intègreront des processeurs ayant des performances plus proches des microprocesseurs utilisés dans les stations de travail.

7.1 Compilateur dynamique

Le Tableau 7.1 donne pour chaque benchmark le temps de synthèse sur Intel et le nombre de tranches qu'utilisent les circuits générés respectivement par Dynabit et ISE (version 9.1) de Xilinx. Dans le cas de Dynabit, le temps de synthèse sur le PowerPC intégré au FPGA (PPC405) est donné. Ce tableau démontre qu'il y a une différence significative entre le temps de synthèse obtenu par Dynabit et le temps de synthèse statique obtenu par les outils standards de Xilinx. L'accélération est entre 750 et 28000 et le temps de synthèse est typiquement en-dessous d'une seconde pour des modules occupant le quart du FPGA (une zone dynamique). Même si on ignore le "temps de démarrage" des outils de Xilinx (en supposant 55 secondes) la synthèse est souvent des milliers de fois plus rapide avec Dynabit. Au niveau du temps de synthèse, ces résultats démontrent clairement que la synthèse à la volée est possible sur des FPGAs présentement disponibles sur le marché.

	Xilinx		Dynabit			Xilinx/Dynabit	
	tranches	temps (sec)	tranches	temps (sec)	temps[1] (sec)	densité	accélération
add1	16	56	16	.002	.029	1.00	28000
mean4	62	63	160	.006	.100	.38	10500
max4	96	61	144	.010	.071	.66	6100
upcase	38	57	112	.040	.074	.33	1425
isqrt-32	585	381	768	.389	8.573	.76	979
bit-count	32	55	320	.010	.141	.10	5500
bit-rev	0	55	480	.014	.280	.00	3928
bit-lg2	22	55	480	.022	.271	.04	2500
fib-16	394	64	256	.014	.303	1.53	4571
fib-32	852	79	512	.031	.594	1.66	2548
fib-64	1782	112	1024	.122	1.387	1.74	918
fib-96	2526	148	1536	.137	2.065	1.64	1080
hash-4	281	64	304	.020	.380	.89	3200
hash-8	581	74	560	.032	.584	1.03	2312
hash-16	1102	109	1152	.135	2.116	.95	881
hash-32	2205	229	2272	.236	3.541	.97	970
hash-48	3455	381	—	—	—	—	—
hashr-4	333	66	432	.070	.174	.77	942
hashr-16	1159	108	1680	.136	1.535	.68	794
hashr-24	1710	166	2544	.216	3.920	.67	768
hashr-32	2229	230	—	—	—	—	—

[1] PPC405 350 MHz intégré au FPGA

TAB. 7.1 – Comparaison de la synthèse entre Dynabit et ISE de Xilinx

Dynabit effectue la synthèse à partir de primitives comparables à un jeu d'instructions 32 bits.

Ce jeu lui permet une correspondance technologique et un placement rapide, mais limite la synthèse à des opérations sur 32 bits. Nous observons avec les applications *bit-count*, *bit-rev* et *bit-lg2* que les instructions produites par Dynabit ne peuvent pas manipuler efficacement les bits à un grain fin. La densité de ces applications est mauvaise lorsque comparée à celle obtenue avec des outils standards de synthèse où il est possible d'exprimer des opérations sur les bits. Ainsi, nous pouvons dire que Dynabit obtient les performances disponibles en matériel mais avec les limites qu'impose la granularité d'un jeu d'instructions.

Néanmoins, les applications numériques ont une densité comparable lorsqu'elles sont compilées à la volée. Par exemple, les applications de cryptographie (hash-*), qui utilisent des étages d'additionneurs et de "xor" afin de calculer une signature, ont moins de 10% de perte de densité. Dans l'optique d'un JIT matériel qui synthétise un circuit à partir d'un programme normalement destiné à exécuter sur un processeur, des performances similaires sont donc anticipées. De plus, nous pouvons voir que les applications de cryptographie (hashr-*), qui utilisent des rotations, perdent jusqu'à 30% en densité afin de diminuer la complexité du routage. Nous verrons à la fin de ce chapitre que la perte de densité est moindre que le gain de densité obtenu par l'évaluation partielle.

Typiquement, la synthèse statique tient compte de d'autres propriétés : la latence, la température et la consommation de puissance. Dans le cas de Dynabit, la température et la consommation de puissance ne sont pas tenues en compte. La latence combinatoire, quant à elle, n'est pas élevée et permet ainsi l'exécution à la fréquence même fréquence que la zone statique (100 MHz) puisque des registres sont ajoutés si nécessaire ; ce qui augmente le nombre de cycles requis pour qu'une donnée traverse l'instruction dynamique.

7.1.1 Phases de synthèse

Afin de mieux cerner les coûts d'exécution, nous avons chronométré les différentes phases de la synthèse sur des benchmarks modérément grands (entre 768 et 2272 tranches). Le Tableau 7.2 contient les temps d'exécution des phases lentes ; les autres phases sont omises puisqu'elles ont un temps d'exécution négligeable.

	Évaluation partielle (sec)	Correspondance technologique (sec)	Placement Place (sec)	Placement Expansion (sec)	Route (sec)
isqrt	.004	.008	<.001	.038	.333
bit-lg2	.001	.001	<.001	.003	.008
hash-16	.004	.002	<.001	.028	.091
hash-32	.009	.005	<.001	.043	.156
fib-64	.008	.004	<.001	.021	.104
fib-96	.012	.009	<.001	.030	.069

TAB. 7.2 – Temps d'exécution des phases de synthèse

Nous pouvons observer l'efficacité du placement linéaire puisque le temps requis pour déterminer une position à chaque primitive est inférieur à 1 ms. La majorité du temps d'exécution est passé dans la création des structures de données nécessaires pour la suite de la synthèse (e.g. expansion des tranches, LUTs, routes, ...).

Comme nous l'avons mentionné précédemment, le routage est la phase la plus coûteuse de la synthèse et domine largement le temps d'exécution des autres phases. Ceci justifie les efforts mis sur son accélération.

7.1.2 Routage

Les différentes passes de l'algorithme de routage ont été séparées puis chronométrées afin d'analyser le travail effectué par celles-ci. Le Tableau 7.3 résume la quantité de routes établies ainsi que le temps d'exécution requis pour chaque passe.

	Routes	Patron		Passe 1		Passe 2		Passe 3		Dijkstra	
	#	#	(sec)	#	(sec)	#	(sec)	#	(sec)	#	(sec)
isqrt	7677	877	<.001	5322	.008	1218	.017	120	.032	140	.177
bit-lg2	1284	292	<.001	945	.007	47	.001	—	—	—	—
hash-16	4515	195	<.001	3773	.004	455	.006	48	.007	44	.072
hash-32	8835	419	<.001	6476	.009	1776	.019	83	.013	81	.079
fib-64	4256	117	<.001	2858	.009	1110	.003	127	.001	44	.089
fib-96	6336	160	<.001	4370	.005	1400	.004	367	.001	39	.053

TAB. 7.3 – Temps d'exécution des phases de routage

Pour éviter les routes diagonales, une passe de routage par patrons applique des routes précalculées et permet un routage efficace de ces routes complexes. Ensuite, une séquence de 3

passes de plus en plus complexes (qui permettent l'utilisation de plus de ressources) sont appliquées. Rappelons que la première passe n'utilise pas les multiplexeurs d'entrées et de sorties, contrairement à la deuxième passe. Puis, la troisième passe contient des règles supplémentaires sur le corps pour des cas fréquents, mais qui nécessitent plus de ressources et qui doivent être appliquées après les premières passes. L'algorithme de Dijkstra est utilisé pour finaliser le routage.

Nous observons que la majorité des routes sont résolues efficacement par les premières passes. La comparaison du temps requis pour résoudre une route nous montre que, pour l'application *isqrt*, la passe 1 est 800 fois plus rapide par route que l'algorithme de Dijkstra. Ceci justifie notre choix de sacrifier la densité pour obtenir un routage plus efficace.

Notons que les algorithmes de routage classiques (e.g. PathFinder) utilisent itérativement l'algorithme de Dijkstra et ne peuvent donc pas obtenir des performances de routage comparables à celles de Dynabit.

7.1.3 Configuration

Nous avons mentionné que la quantité de données générées par un compilateur dynamique vers un jeu d'instructions et celles générées par la synthèse à la volée n'est pas du même ordre de grandeur. Le Tableau 7.4 démontre qu'un nombre élevé de points de configuration doivent être programmés afin d'implanter le module dynamique. Notons par exemple que l'application *isqrt* ne contient pas plus d'une centaine d'instructions assembleurs tandis que Dynabit doit générer environ 40 000 points de configuration dont chacun affecte entre 1 et 16 bits de configuration.

	CFGs	PIPs	Configuration PPC405 (sec)
add1	288	505	.026
isqrt	14400	24581	.745
bit-lg2	3072	3921	.191
hash-16	7392	13206	1.461
hash-32	14112	26661	1.793
fib-64	6624	10860	.399
fib-96	9696	15742	.510

TAB. 7.4 – Quantité de points de configuration et temps de configuration

La quantité de bits modifiés engendre un coût de configuration non négligeable. Il est à noter

que la modification d'un bit nécessite des accès mémoire sur le bus PLB, suivis de transactions sur le bus OPB qui ne sont pas en mode rafale. Par conséquent, le temps de configuration est présentement élevé mais peut être amélioré par l'écriture d'un module matériel mieux adapté à la synthèse rapide que le module HW_ICAP fourni par Xilinx.

De plus, notons que le ICAP communique à 33 MHz avec des entiers de 8 bits et doit nécessairement communiquer une trame (*frame*) complète (80 CLBs pour le VP30). Dans le but de répondre aux besoins de la reconfiguration dynamique, Xilinx a amélioré, sur le Virtex-4, la communication avec le ICAP en augmentant sa fréquence à 50 MHz et en communiquant des entiers de 32 bits tout en diminuant considérablement la granularité des trames (8 CLBs). Le temps de configuration pourrait être réduit considérablement.

7.2 Interface

Le processeur communique avec la zone dynamique au travers d'une interface mémoire qui supporte le mode de communication en rafale (*Burst Mode*) permettant de communiquer rapidement des groupes de données. La taille des blocs envoyés à la zone dynamique est une ligne de cache, soit 8 entiers de 32 bits.

Fonction	Temps 1000000 données (sec)	PPC 350 MHz (cycles/donnée)	PLB 100 MHz (cycles/donnée)
map	9.316	3 260	931.6
#%map	4.124	1 443	412.4
combine1	.126	44.1	12.6
combine2	.208	72.8	20.8
combine3	.369	129.5	36.9
combine4	.391	136.9	39.1
brute	.058	20.3	5.8

TAB. 7.5 – Bandes passantes des communications pour différentes fonctions d'interface matérielle

Le Tableau 7.5 fait la comparaison des coûts de communication selon les différents mécanismes utilisés. Les temps sont mesurés sur le PPC405 du FPGA. Nous avons calculé le nombre de cycles en temps absorbé pour traiter une donnée résultante en fonction de la fréquence du processeur et de son bus.

Dans notre implantation, une fonction encapsule les mécanismes de communication afin de

les cacher. Ainsi, l'appel à cette fonction effectue les communications requises pour transférer une donnée. La fonction `map` de Scheme effectue pour chaque élément d'une liste chaînée un appel à cette fonction qui doit à son tour appeler une fonction en C. Cette cascade d'indirections ajoute des coûts de communication non négligeables et rend cette interface peu attrayante.

La fonction `#%map` est une implantation native de la fonction `map` qui permet d'éviter le coût des appels répétitifs à une fonction native. L'inconvénient de cette fonction est qu'elle envoie séquentiellement, une à la fois, les données et ne tire pas profit du mode *pipeline* disponible. Quoique plus rapide, ses performances démontrent que cette fonction n'absorbe pas les coûts de communication.

Les fonctions `combine-*` envoient nativement les données à partir de vecteurs de données 32 bits. Ce type de communication est adapté au mode *pipeline* et possède des coûts de communication raisonnables. Nous avons observé que les transactions mémoires du processeur par le PLB ne sont pas aussi efficaces que nous l'espérions. Comme la quantité de données à traiter est élevée, la majorité des accès ne sont pas en cache et nécessitent des transactions sur le PLB. La fonction `combine1`, qui applique une fonction à tous les éléments d'un vecteur pour produire un vecteur des résultats, doit effectuer 4 transactions sur le PLB : lecture de la donnée en mémoire, écriture et lecture de la donnée sur le contrôleur dynamique, puis écriture en mémoire du résultat. Ainsi, dans le meilleur cas, le *pipeline* fonctionne en-dessous de 1 cycle sur 12, ce qui est bien en-dessous de ses performances potentielles.

Notons que la complexité et la latence des instructions dynamiques n'influencent pas la vitesse de communication puisque les communications sont en mode *pipeline*. Ainsi, les gains de performance d'une instruction dynamique sont proportionnels à la quantité d'opérateurs de base constituant la fonction compilée.

Pour démontrer l'effet des communications sur le PLB, nous avons implanté une fonction native (`brute`) qui envoie une séquence de nombres consécutifs et retourne le premier nombre pour lequel l'instruction spécialisée retourne "vrai". Le but est d'analyser l'envoi de données sans aucune lecture ou écriture en mémoire externe. Ce type de communication est utile pour les applications comme MD5 qui effectue une recherche exhaustive (*brute force*) sur un intervalle. Le coût des communications baisse à 5.8 cycles, soit environ 3 cycles par donnée communiquée (écriture puis lecture).

Nous pouvons conclure que le processeur n'est pas en mesure de fournir assez rapidement

les données aux instructions dynamiques et que le couplage n'est pas assez rapproché pour permettre une utilisation en mode *pipeline*. Nous proposons dans le prochain chapitre des solutions alternatives pour absorber les coûts de communication.

7.3 Évaluation partielle

Dans cette section, nous décrivons des applications qui peuvent bénéficier de l'évaluation partielle et de l'accélération matérielle. Nous voulons ainsi justifier le modèle d'exécution que nous proposons.

7.3.1 Générateur de nombres pseudo-aléatoires

Un générateur de nombres pseudo-aléatoires (e.g. PRNG), qui utilise la congruence linéaire, se définit comme suit : $x_n = (Ax_{n-1} + C) \bmod M$ où A, C et M sont des constantes prédéfinies. La Figure 7.1 contient le code source d'une fonction qui retourne un générateur de nombres pseudo-aléatoires selon les constantes a, c et m.

```
1 (define make-prng
2   (lambda (a c m)
3     (lambda (x)
4       (modulo (+ (* a x) c) m))))
```

FIG. 7.1 – Code source de PRNG

Typiquement, la multiplication et le modulo sont des opérations coûteuses qui peuvent s'implanter en matériel. Rappelons que la taille d'un circuit calculant une multiplication et un modulo s'exprime par une équation quadratique en fonction du nombre de bits calculés. En connaissant l'implantation des opérateurs en matériel, nous pouvons borner inférieurement la taille du module générique à au moins 96 primitives en ne tenant pas compte des décalages (sachant qu'un multiplicateur 32 bits a 32 additionneurs conditionnels et que le modulo a 32 comparateurs et 32 additionneurs conditionnels).

	Paramètres dynamiques			primitives	tranches	temps
	A	C	M			(sec)
mod1	0x14DE57	0xA8C31F	0x1F212C45	30	480	.022
mod2	0x14DE57	0xA8C31F	0x1024	63	1008	.172
mod3	0x14DE57	0xA8C31F	0x1000	23	368	.017
mul1	0x1000	0xA8C31F	0x1F212C45	10	160	.008
mul2	0xCCFF00FF	0xA8C31F	0x1F212C45	27	432	.032
mul3	0x357BACDE	0xA8C31F	0x1F212C45	39	624	.048
pire	0x55555555	0x1	0x13	95	1520	.144
meilleur	0x1000	0	0x1000	2	32	.004

TAB. 7.6 – Évaluation partielle de générateurs de nombres pseudo-aléatoires (PRNG)

Le Tableau 7.6 contient la densité et le temps de synthèse sur Intel de différents générateurs selon différentes constantes. Nous observons que le pire cas reste inférieur à 96 primitives (la taille du module générique). Nous tenons compte des décalages dans le calcul de nos primitives puisqu'ils utilisent un espace physique.

Les constantes influencent considérablement la taille du circuit généré. Notons que pour un modulo, la quantité de soustracteurs conditionnels est inversement proportionnelle à la quantité de bits significatifs de la constante. Comme nous l'avons démontré, la multiplication par une constante se simplifie par les optimisations de Rice. Le pire cas que nous ayons produit est une séquence alternée de 0 et de 1. Il est à noter qu'une séquence de 1 est remplacée par un décalage et un soustracteur.

La Figure 7.2 contient le code après l'évaluation partielle. Nous y observons la multiplication par une constante (e.g. tmp.13), l'addition par une constante (e.g. tmp.14) suivie d'une cascade de soustracteurs conditionnels (additions conditionnelles du complément à deux) pour effectuer le modulo.

```
1  (lambda (x.13)
2    (let* ((tmp.13 (#%add
3                    (#%add (#%add (#%shl x.13 16)
4                                  (#%shl (#%neg x.13) 13))
5                           (#%add (#%shl (#%neg x.13) 9)
6                                  (#%shl x.13 6)))
7                    (#%add (#%add (#%shl x.13 4)
8                                  (#%shl x.13 3))
9                           (#%add (#%neg x.13)
10                                 (#%add (#%shl x.13 20)
11                                        (#%shl x.13 18))))))
12           (tmp.14 (#%add tmp.13 11059999))
13           (tmp.15 (#%cadd (#%le 4178141736 tmp.14) tmp.14 116825560))
14           (tmp.16 (#%cadd (#%le 2089070868 tmp.15) tmp.15 2205896428))
15           (tmp.17 (#%cadd (#%le 1044535434 tmp.16) tmp.16 3250431862))
16           (tmp.18 (#%cadd (#%le 522267717 tmp.17) tmp.17 3772699579)))
17      tmp.18))
```

FIG. 7.2 – Évaluation partielle sur PRNG (a=0x14DE57 c=0xA8C31F m=0x1F212C45)

L'ensemble des opérateurs de base ainsi produit est transformé en un ensemble de primitives matérielles puis configuré sur le FPGA pour obtenir l'instruction spécialisée présente à la Figure 7.3.

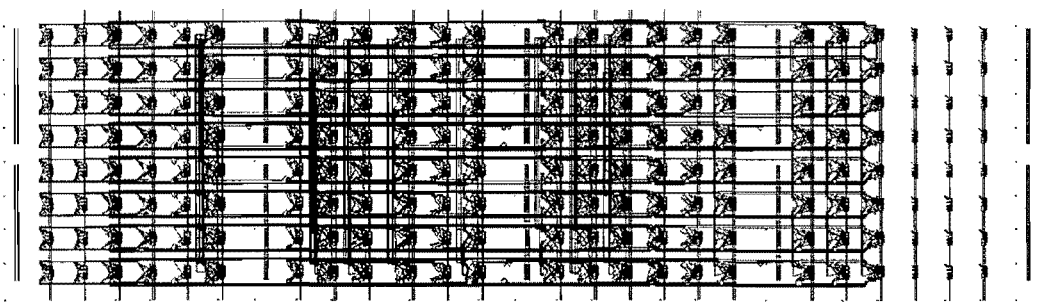


FIG. 7.3 – Capture d'écran de PRNG dans FPGA_Editor

7.3.2 Cryptographie MD5

L'algorithme de cryptographie MD5 est utilisé pour produire une signature (*hash*). Nous avons expliqué à la Section 6.2.2 comment implanter l'algorithme MD5 en utilisant l'évaluation partielle pour que son exécution soit à la fréquence maximale du FPGA.

Malheureusement, la taille de l'algorithme est trop élevée pour tenir dans une zone dynamique. Il doit être séparé dans les deux zones. Or la communication entre les deux zones n'est pas implantée et, par conséquent, notre système n'est pas en mesure de supporter complètement

l'algorithme.

Afin de mesurer les performances de notre compilateur, nous avons compilé le plus grand nombre d'étages de l'algorithme entrant dans une zone dynamique (28 étages sur 64) en 493 ms, ce qui nécessite 2112 tranches. La Figure 7.4 montre la capture d'écran de ce qui est produit par le compilateur.

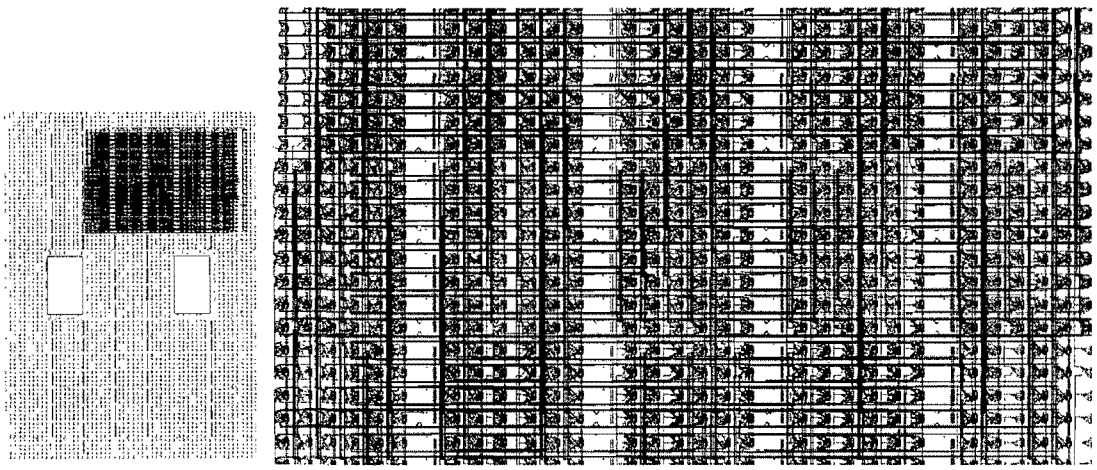


FIG. 7.4 – Capture d'écran de MD5 dans FPGA_Editor

Notre compilateur n'est pas en mesure de continuer le placement vers le bas puisqu'un processeur (inutilisé) est présent à droite. Les nouvelles versions des FPGAs de Xilinx positionnent les deux FPGAs du même côté de la puce afin d'éviter ces cas. Ainsi, nous pouvons affirmer que MD5 tient dans les nouvelles puces (Virtex4) en utilisant la synthèse à la volée et permet l'exécution à la fréquence maximale.

Dans le cas de recherche exhaustive, la génération des entrées doit provenir du matériel pour obtenir les performances maximales. Le prochain chapitre propose des solutions à ce problème.

Une implantation matérielle efficace a été réalisée sur un VirtexII (XC2V4000-6) et démontre qu'il est possible de produire une configuration pour MD5 qui occupe 7997 slices dont le circuit a un pipeline de profondeur 64 et permet l'exécution à une fréquence de 93.4 MHz [JTS05].

7.3.3 Autres applications

De nombreuses autres applications peuvent tirer profit de notre modèle d'exécution.

Notons par exemple l'entraînement de réseaux de neurones. Un neurone multiplie chacune de ses entrées par un poids constant et calcule la somme de ces résultats. Il est possible d'effectuer un entraînement par lot (*batch*), en accumulant les correctifs jusqu'à la fin d'un lot. Ainsi, cette application pourrait produire des instructions spécialisées qui implémentent les neurones et modifier dynamiquement ces instructions lorsque les poids sont mis à jour. Il a déjà été démontré rentable d'entraîner un réseau de neurones en effectuant des reconfigurations dynamiques d'un FPGA [EH94b].

Un autre exemple d'application est le traitement de modèle 3D où le modèle est un ensemble de vecteurs à 3 dimensions : x, y et z. En 3D, les transformations affines passent par la coordonnée homogène (vecteurs à 4 dimensions). Ainsi, une transformée affine consiste à multiplier ces vecteurs par une matrice 4x4 constante. Il est possible d'ajouter une fonction qui effectue en matériel la multiplication de matrices avec une instruction de 4 entrées vers 4 sorties.

Comme nous le voyons, il existe plusieurs applications qui bénéficieraient de l'accélération matérielle. Elles nécessitent toutes des mécanismes de communication différents mais elles ont un point en commun : la synthèse à la volée est nécessaire.

7.3.4 Conclusion

Les temps de synthèse obtenus par le compilateur Dynabit sont en-dessous de la seconde et démontrent ainsi qu'il est possible de produire à la volée des configurations pour un VirtexII-Pro de Xilinx.

Nous avons observé que le rapport de densité entre les circuits statiques et dynamiques ne démontre pas de différences significatives lors de la compilation d'applications destinées à être exécutées sur un processeur. Une légère perte de densité est observée et provient du fait que Dynabit applique des transformations au circuit afin de simplifier la complexité du routage et ainsi diminuer ses temps de compilation.

De plus, l'utilisation de l'évaluation partielle permet de réduire considérablement la taille d'un circuit en supposant constant certaines variables de l'application. Dans le cas de certaines applications, cette réduction dépasse largement la perte de densité engendrée par la compilation à la volée.

Avec l'analyse des temps d'exécution des différentes phases de la synthèse à la volée, nous pouvons observer que la majorité du temps de synthèse est passé dans la phase de routage. De plus, nous pouvons observer que la majorité des routes sont résolues efficacement et que certaines routes plus complexes nécessitent la majorité du temps de routage.

Sur le FPGA que nous avons utilisé, les temps de reconfiguration ne permettent pas un changement rapide de contexte. Par conséquent, le coût de reconfiguration doit être absorbé par l'utilisation répétitive du module dynamique. Nos techniques de synthèse rapide ont permis d'obtenir des temps de synthèse minimes et similaires aux temps de reconfiguration qui étaient auparavant négligeables. Nos résultats démontrent que la synthèse rapide n'est plus l'élément limitatif à la construction d'un compilateur dynamique (JIT).

Pour qu'une application obtienne de bonnes performances, le couplage entre le processeur et la zone dynamique doit permettre une interface de communication avec une courte latence et une bande passante élevée.

Chapitre 8

Synthèse de haut niveau

La synthèse de haut niveau consiste à déterminer comment implanter en matériel les concepts d'un langage de programmation. Ce domaine est étudié par plusieurs groupes de recherche qui effectuent de la compilation statique d'un langage de programmation, tel que le langage C [Cal02, TGP07], vers du matériel.

Le début de notre recherche s'est orienté sur l'étude des techniques de compilation statique vers le matériel. Ainsi, nous avons contribué au développement du compilateur CASM [DB04b, DB04a], un compilateur de machines à états algorithmiques synchronisées par les données. Ce langage cible les développeurs avec peu d'expérience en matériel. Il est sûr et ainsi empêche l'écriture de forme générant des erreurs en matériel (e.g. boucle combinatoire, *clock skew*, ...). CASM est un langage de choix pour le développement de prototypes puisqu'il permet d'exprimer avec facilité des concepts typiquement complexes et fastidieux à implanter.

Les propriétés de CASM nous ont permis d'étudier les techniques de compilation d'un langage de haut niveau vers le matériel. Nous avons réalisé des prototypes afin de déterminer les différentes approches possibles pour implanter en matériel certains concepts présents dans les langages de haut niveau. Nous avons réalisé un prototype d'un compilateur du langage Scheme vers le matériel [BSMFD05] qui, suite aux travaux de Xavier Saint-Mleux, est devenu le compilateur SHard [SMFD06]. L'article qui suit, présenté à *RSP2005 : Rapid System Prototyping*, introduit les techniques de génération de matériel pour implanter les concepts de haut niveau tels que les appels de fonction et les fermetures qui sont utilisées dans SHard.

High Level Synthesis for Data-Driven Applications

Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley, Jean Pierre David
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

***Abstract*— John von Neumann proposed his famous architecture in a context where hardware was very expensive and bulky. His goal was to maximize functionality with minimal hardware. Presently, logical gates are nearly free and single chips will soon contain billions of gates. However, most current designs are still based on Von Neumann's architecture because processors are built on this model. Nevertheless, the main current challenge is to be able to design, refine, synthesize and verify new architectures in a minimum time and with a maximum computational performance regardless of the gate count. Data driven architectures enable a high level of parallelism because instead of a single controller managing all the resources (and often a single ALU), tens or hundreds of small controllers can now operate in parallel on local processing units. This paper presents an environment for the high level description, refinement, synthesis and verification of such systems. Our own HDL is presented with its compiler and we show how it can be used as the intermediate language of a compiler for an even higher level functional programming language. Ongoing work will enable the interfacing with other languages (from both hardware and software communities). We also intend to target asynchronous designs.**

I. Introduction

John von Neumann imagined a new way to program the ENIAC machine in 1948 : storing a program in ROM and letting the machine execute the now well known fetch-decode-execute cycle. Thanks to this improvement, he reduced the programming time from several days to a few hours. It was the starting point of a long era of control dominant architectures, which still lasts today.

A new approach was introduced in the 70's by Dennis [1], who proposed the first token machine, an architecture where the operations are triggered by data instead of a central controller managing all the computations. Data move on a graph in the same way as tokens in a Petri net. This architecture theoretically offers the highest degree of parallelism achievable because the operations start as soon as data are ready. The main bottleneck was the implementation of data matching which is required before a (multiple-operand) operation is executed.

Both architectures have evolved but the superscalar processor exceeded token machines in the 90's by integrating the token machine approach but on a restrained data set. More generic than token machines and also much more pervasive in the industry, the central control won this battle.

The present technological context is however completely different compared to 1948. Billions of transistors consuming a few Watts are available in a single chip. Reconfigurable computing allows the implementation of new architectures within seconds. A chip can run several years before encountering a failure. The main challenge now is to be able to manage the implementation of so many resources in a proven safe way and quickly obtain maximum computation efficiency and minimum power consumption.

The major alternative to the processor is the design of dedicated chips in ASIC or FPGA. VHDL [2] and Verilog [3], which were developed in the 80's, continue to be the most widely used HDL in the industry even though they are low level languages. Significant work has been done to develop higher level languages capable of targeting hardware : HandleC [4], HardwareC [5], Transmogrifier C [6]. But C-like languages have complex semantics that make it difficult to formally prove the correctness of a design. Another approach aims to model the design at a higher level. SpecC [7], SystemC [8], SystemVerilog and eSys.Net [9] offer this kind of high level environment. But once validated, these designs must be refined, often manually, to get equivalent RTL code.

We believe that functional languages can take an important role in current hardware design because they are high level and safe languages, and are based on the same concepts as token machines and so can in principle lead to highly parallel architectures. But this is only achievable if the architecture is built from the algorithm description instead of trying to adapt the algorithm to fit a given architecture as was done up to now. Various approaches have already been presented in previous work (Confluence, Lava [10], Hydra [11]). To our knowledge, this is the first implementation allowing fully general recursion. In this paper, we describe an approach to automatically synthesize an algorithm described in a functional language by successive refinements.

The paper is organized as follows: Section II describes our new version of the CASM language and compiler, which will be used as an intermediate representation of a token machines. Section III proposes a methodology to compile a functional language into a specific CASM-based representation of token machine. The transformations are illustrated by examples throughout the whole document. Our conclusions and future work are presented in Section IV.

II. CASM Language

The CASM language is designed to ease the development of hardware components and applications by people with little or no knowledge of digital circuits. The design philosophy of CASM is to provide a safe language where the execution does not deviate from the language semantics. Traditional

HDLs allow a user to describe circuits whose behavior may become unstable or unpredictable under special circumstances. Glitches, gated or synthesized clocks inducing delays, interfaces between multiple clock domains, combinatorial loops and so on can induce non-deterministic behavior. CASM does not let programmers make this kind of (synthesizable) errors. All programs written in CASM are synthesizable, fully predictable and their execution conforms to the simulation.

Twenty years ago, programmers had to be conscious of the hardware because it was a major limitation in memory and time. A programmer had to optimize his assembly code to increase the speed and reduce the memory requirements as much as possible. Nowadays, most programmers are used to thinking about an algorithm as a flow chart or a finite state machine but very few of them are aware of what happens at the gate or RT level. Most existing HDL serve to describe the actual physical components and connections of the circuitry instead of presenting the algorithmic concepts. But it is difficult to implement these concepts in a common HDL without spending time understanding all the subtleties and physical limitations of hardware. These problems should not be the concern of programmers who are interested in speedup of a hardware implementation.

Algorithmic State Machines (ASM) charts provide an easy and intuitive way to describe small algorithms. Figure 1 represents the chart for Euclid's GCD algorithm. It is easy to understand the behavior, state by state, of the algorithm and thus understand how the problem is solved. Nevertheless, a graphical interface is impractical for large algorithms and is hard to manage.

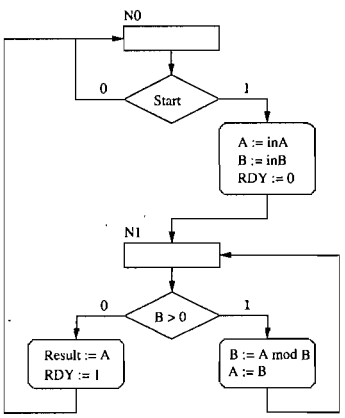


Fig. 1. ASM Chart : Euclid's GCD

To write this algorithm in a standard HDL, programmers must handle the interface of the components and implement a protocol to accept and confirm the input's reception (`inA`, `inB`) and `Result` transmission. Programmers must also correctly handle state transitions from state `N0` to state `N1` so that it only happens when data have been received at both inputs. The return to state `N0` must only occur when the result has been accepted by the outside world.

CASM is an intermediate level language of higher level than RTL (VHDL/Verilog) but which remains at a lower level than languages used for software development (C/Java). Our goal is to obtain a cycle accurate language that allows programmers to implement efficient algorithms with little formal training in CASM.

We based the semantics of our language on state machines. CASM is a textual representation of state machine charts augmented with higher level synchronization, communication, storage and recursion features, which will be described in the following sections. Because hardware can be modeled as a collection of parallel connected state machines (Mealy/Moore) [12][13], our language can be used to describe almost all sequential circuits including multiple clock designs.

```
1. input inA{protocol="FS"}[32];
2. input inB{protocol="FS"}[32];
3. output result{protocol="FS"}[32];
4.
5. asm {
6.   register A[32] = 0, B[32] = 0;
7.
8.   N0: A := ina; B := inb;
9.       goto N1;
10.
11.  N1: if (B>0)
12.          B = A mod B;
13.          A = B;
14.          goto N1;
15.      else result := A;
16.          goto N0;
17.      end;
18. }
```

Fig. 2. CASM Example : Euclid's GCD

Figure 2 shows the implementation of Euclid's GCD algorithm in CASM. The reception of inputs is specified on line 8 in state `N0`. Inputs are received with a full-synchronization protocol (`FS`). The transition to state `N1` occurs when all transfers required by state `N0` are done. Due to the "`result := ...`" on line 15, the receiver must also be ready to receive the result before the system can go back to state `N0`. While the implementation of protocols and conditional transitions are cumbersome in typical HDL, they are much easier in CASM.

The State Machine is a simple conceptual paradigm to represent an algorithm but it is still not expressive enough to easily be used as a programming language capable of describing large and complex components. CASM has builtin features that express frequently used high level concepts; these concepts are explained below.

Model

A CASM program is a collection of devices connected together by synchronized channels. The behavior of these devices is described by state machines. Every CASM file contains a device description and may contain instances of other devices. Figure 2 describes the behavior of the GCD device. The GCD device contains two registers that are explic devices and an implicit state register. It also contains inpu and output channels and combinatorial operators. The model corresponding to this description is shown in Figure 3. A

transaction is a data transfer from a given source to a given target. They are activated by the transaction controller and actually occur when both source and target are ready to complete.

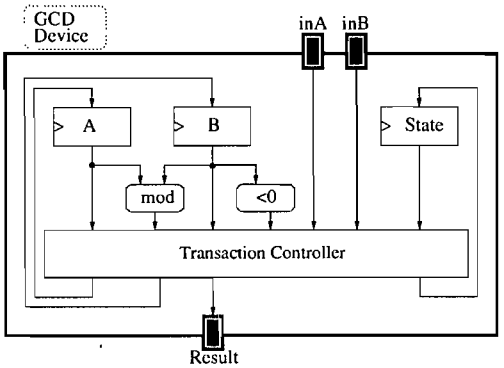


Fig. 3. Model of Euclid's GCD

The square boxes represent device instances, the rounded boxes combinatorial operators and the gray boxes device interfaces. Arrows are channels that contain both data and synchronization signals.

Connections and Transfers

Our language has two kinds of assignments: connections and transfers. Connections use the = symbol and transfers use the := symbol. A connection physically connects a source to a target enabling data transmission implemented by a predefined protocol. A transaction may occur each clock cycle. Transfers are a restricted version of connections where only a single transaction can (and must) occur before a jump to the next state is processed. In Figure 2, a connection can be found on line 12 and a transfer can be found on line 8.

Semantics

Each application written in CASM has an implicit `clk` and `nreset` signals that are used by devices and state machines. When the `nreset` signal is low, the component does a global reset. Steps are :

1) Initialize the devices with initial values.
2) Move all state machines to their initial states.
3) Wait until the `nreset` signal is deasserted.

In Figure 2, the registers will take the initial value of 0 (line 6) on a global reset and the state machine will move to the first state (`N0`).

When an event occurs on the `clk` signal, all the activated transactions are processed in parallel. The current state may be updated if a jump was also activated. From this new global state, the following evaluation steps are performed in preparation of the next clock event.

1) Evaluate conditional expressions and activate selected transactions.
2) Evaluate R-values and completion logic in parallel.
3) If all activated transfers can complete during current clock cycle, activate the jump to the next state.
4) Wait until next clock event.

In Figure 2, when in state `N1`, the conditional expression is evaluated to determine which of the two connections that continue the calculation (line 12) or the transfer that produce the result (line 15) must be activated. When the transfer is activated, the state completion condition depends on the readiness of the receiver. The transaction will not complete until the result is consumed. When the receiver is ready, the transaction and the jump are both activated to immediately move to state `N0` at the next clock event.

Self-synchronized data transfers

Data transmissions are done through channels using standard protocols. The channel contains implicit signals to synchronize both connected devices. The available standard protocols are:

- **Full-Synchronization** *(FS)* The transaction occurs only when both receiver and sender are ready.
- **Half-Synchronization** *(HS)* The transaction occurs when the sender is ready. Data is lost if the receiver is not ready.
- **No Synchronization** *(MAIN)* The transaction always occurs.

To avoid breaking transmission invariants and data loss, stronger protocols cannot be connected to weaker protocols. For example, an FS channel cannot send its data through a HS channel.

Extended Expressions

Expressions have an elaborate type system that includes protocol and variable widths. The type system enforces protocol (full-synchronized data is definitely consumed and never lost). Moreover, no precision is lost in arithmetic calculation without explicit programmer input, even if complex expressions are evaluated. The compiler infers the width through the expressions.

Protocols are automatically inferred through expressions. As an example, to perform the sum of two full-synchronized inputs, the system deduces that the two terms must be present.

CASM provides functions that simplify the implementation of data-driven applications. Functions like `queue`, `buffer` and `pipeline` are used to implement pipelined expressions. The expressions `reg = pipeline{cycle=4}(a*b);` and `reg = queue(queue(buffer(c)-d) >> e);` are pipelined calculations. A queue acts as a pipeline register between two stages. Queue is a synchronized component and it automatically stalls when needed. A buffer consumes inputs when the sender is ready and thus frees the sender. It sends the data to the receiver as soon as it is ready, possibly in the same clock cycle. The pipeline function automatically produces a pipeline with its expression. The buffer size and the pipeline depth are parameterizable options.

Recursion

To increase the expressive power of our language, recursion is allowed. In addition to jumps, one can use the `call` instruction to branch to another state and keep a reference

of the return point of the subfunction. It resembles the JAL (Jump and link) instruction. The return state is the return point. In Figure 4, on line 12, there is a call to the `fact` state and the return point is the `send` state. The return, which is a branch to the first return point on the stack, occurs on line 15.

```
1. input n{protocol="FS"}[8];
2. output r{protocol="FS"}[32];
3. queue qn{size=16}[32], qr{size=4}[32];
4.
5. always { qn = n; r = qr; }
6.
7. asm fact{depth=1<<(n.width-1)} {
8.     stack args{size=fact.depth}[8];
9.     register p[32];
10.
11.    receive: p := qn;
12.             call fact; return send;
13.    send:    qr := p; goto receive;
14.
15.    fact:    if (p==0) p = 1; return;
16.             else args := p;
17.               if (fact.complete)
18.                 p = p-1;
19.               end;
20.               call fact; return mul;
21.             end;
22.    mul:     p := args * p; return;
23. }
```

Fig. 4. CASM Example : Buffered Recursive Factorial

Recursive state machines must specify a stack depth because it is not possible to statically bound it in the general case. When state machines are not recursive, the language states that the stack depth is guaranteed to be large enough and programmers can safely omit this information.

Anticipative transaction evaluation

CASM can determine whether a transfer has already completed or can complete in the current clock cycle. The completion information can be accessed using the `complete` attribute. When associated to a state name, the meaning is that the jump is activated and will occur at the next clock event. This is illustrated in Figure 4, line 17 where `p` must not be modified before being pushed in the stack `args`. So when in state `fact` and `(p!=0)`, the state machine waits until the queue `args` is ready to receive new data. As soon as the condition is met, the jump and the connection between `p` and `p-1` are activated in order to complete all transactions in a single clock cycle. When a transfer is conditional on another `complete` signal, the system becomes anticipative. This is illustrated by the example in Figure 5.

```
1. if (t1.complete) t2: a:=x; end;
2. if (t2.complete) t1: b:=y; end;
```

Fig. 5. Anticipative transaction example

Transfer `t2` is activated only if `t1` completes in the current clock cycle but transfer `t1` is activated only if `t2` completes in the current clock cycle. There are theoretically two possible solutions : `t1` and `t2` complete together or `t1` and `t2` do not complete. The resolution of such ambiguities is processed by considering that *a-priori* all transactions complete. When this is impossible, conflicting transactions are iteratively deactivated until a solution is found. It may happen that no solutic exists, or can be found by this heuristic, which generates an error at compilation time.

Generic Devices

Generic devices have predefined behavior and synchronized interface but nothing is stated about their implementation. Our system provides parameterizable generic devices for commonly used components such as memories, stacks, and queues. These components are so widely used that we provide a syntax that easily fits in a CASM program and different implementations. The system also allows users to provide an other implementation that better suits their requirements.

For example, in Figure 4 queues are declared on line 3. These queues will be used to buffer inputs and outputs. Inputs are received and added to the queue on line 5. Since the inputs are queues, the sender does not block when transferring its data. When a buffer is full, the sender is blocked until there is some space in the queue. An example of input consumption can be found on line 11.

A special syntax has been created to easily interface with these components. For example, the array operator is used to bind an expression to an address channel. So, one can use the `x := mem[expr];` syntax to read something from memory and `mem[expr] := expr;` to write something to memory.

We also provide specific implementations for different FPGA technologies. Some FPGAs provide embedded memories, CPUs, or others components. Users can configure the behavior and the chosen implementation using parameters or let the compiler choose appropriate parameters.

III. Compiling Functional Programming Languages to CASM

An interesting application of the CASM language is as an intermediate language for even higher-level hardware-description languages. It is particularly attractive to use CASM's high-level synchronization mechanisms for expressing data-flow architectures. Consequently we have begun investigating the use of the CASM language for implementing a compiler for parallel functional programming (FP) languages. Given that we are in an early exploratory stage, we have chosen to use a small FP language with few builtin operators and a parenthesized prefix syntax based on the Scheme programming language [14], which is both easy to parse and extend if the need arises. Currently our FP language is purely functional (no side-effects are possible), lexically-scoped (variables refer to the closest enclosing declaration in the program source code), and strict (arguments are evaluated before the function is called).

There are several reasons why an FP language makes a good HDL. The formal semantics of an FP language is typically smaller and cleaner than that of an imperative language suc as C and Java. This is helpful for verifying the correctnes of programs and also provides a solid framework for the program transformations and optimizations that are performed

by the compiler, such as function inlining, loop unrolling, code motion, partial evaluation, and so on. FP languages are declarative in nature, which means that the source code is closer to a specification than an implementation. It is thus more likely that a subsystem designed for one application can be reused in another application with little or no change. The absence of side-effects makes it easy to exploit parallel execution. Finally, given the existence of compilation techniques for compiling FP languages to efficient machine code, it seems that a single FP language could be used as an approach for hardware/software co-design. Only one language needs to be learned and used by the designer, and the partitionning of the system into hardware and software subsystems can be done by the compiler with some assistance from the designer (through source code annotations) or possibly fully automatically based on design constraints (silicon area, processing throughput, etc).

Recursion and Higher-Order Functions

A fundamental problem in compiling a parallel FP language to hardware is the handling of recursion and higher-order functions (functions which take functional arguments or that return functions). These features are necessary for full support of the FP programming style. To be consistent with our goal of providing a high-level programming style we would like to avoid placing restrictions on the type and depth of recursion and the use of higher-order functions. A (practically) unrestricted recursion depth can be achieved using a single large memory to store a stack of call frames for each process. The problem with this approach is that the memory would be a bottleneck that limits the system's parallelism. Moreover the long access latency of a large memory will reduce the performance of recursion. Our approach is to use one small memory for each function return point. This way the access time is short and different processes can execute in parallel as long as they are executing different sections of the program. Memory fragmentation due to alignment is eliminated because the width of the memory is equal to the size of the frame associated to that particular return point. To allow several processes to coexist the frames are explicitly linked in a chain rather than using a contiguous stack representation.

Closures

The traditional representation of lexically-scoped functions is the closure; a piece of data containing the value of the function's free variables and a reference to the implementation of the function's abstraction (i.e. where the free variables are viewed as additional parameters). Through the continuation-passing-style (CPS) transformation [15], a functional program is converted to a form where function returns are emulated using function calls. After a CPS conversion, all function calls and returns correspond to tail function calls (for which no stack frame needs to be pushed on the stack). Each non-tail function call in the original program is transformed into a tail function call to the function with an additional "continuation" parameter, which is a closure that contains the values that are needed at the return point by the calling function. A tail function call to the continuation closure corresponds to returning a result to the return point. Thanks to CPS conversion, the implementation of recursion and higher-order functions boils down to the implementation of tail function calls and closure allocation/deallocation.

In hardware, a CPS converted function is a device that receives requests through an input channel. A tail function call corresponds to sending a request containing the parameters to the input channel of the device that implements the function. A device typically performs some internal computation on the parameters received and then sends a request to some other device (either to return a result or perform a non-tail call of the source program). Our implementation of closures consists in adding a small memory local to a device (the closure memory), wide enough to hold its free variables. A closure reference is the address of the free variables in this memory and some tag bits that identify the device that implements the function. The tag is used in the implementation of tail function calls to closures to direct a request containing the parameters and the closure's address to the appropriate device for this closure. Upon receiving such a request the device uses the closure address to lookup the free variables in the closure memory and combine them with the device's other parameters to perform the computation appropriate for this function. Note that a function with no free variables does not need a closure memory, but the tag bits are needed to distinguish it from other closures.

Memory Management

An entry in a closure memory must be allocated when a closure is created. All free entries are linked in a free list to simplify the allocation mechanism. When the closure is no longer needed for the computation it must be deallocated to make room for future closure creations. This is done by adding the deallocated entry to the free list. This operation could be performed with a garbage collector, like in software implementations of FP languages. However this is hard to implement in hardware, especially with several distributed closure memories. Instead we perform automatic deallocation of continuation closures (the closure entry is freed when the closure is called) and rely on explicit deallocation operations in the program for non-continuation closures.

An example: factorial

To illustrate the compilation of recursive FP programs to CASM, we use the factorial function. The input code supplied to the compiler is shown in Figure 6. The factorial function itself is defined on line 1 while line 4, which is the program's body, is a call to this function with a free variable (n) as an input. The resulting circuit will have a single input channel for the n parameter and a single output channel for the result, both full-synchronized. Annotations may be added to specify, for example, n's width in bits.

The input code is first CPS converted, as described above. The next phase does a combined control and data flow analysis

```
1. (define fac
2.   (lambda (x)
3.     (if (= x 0) 1 (* x (fac (- x 1))))))
4. (fac n)
```

Fig. 6. Factorial function, as input

(0-CFA) [16], which attaches an abstract value to every variable that might refer to a function. Thus, every function call is annotated with a list of all functions it might jump to. Finally, the annotated code goes through closure conversion [17], which makes a closure's free variables explicit.

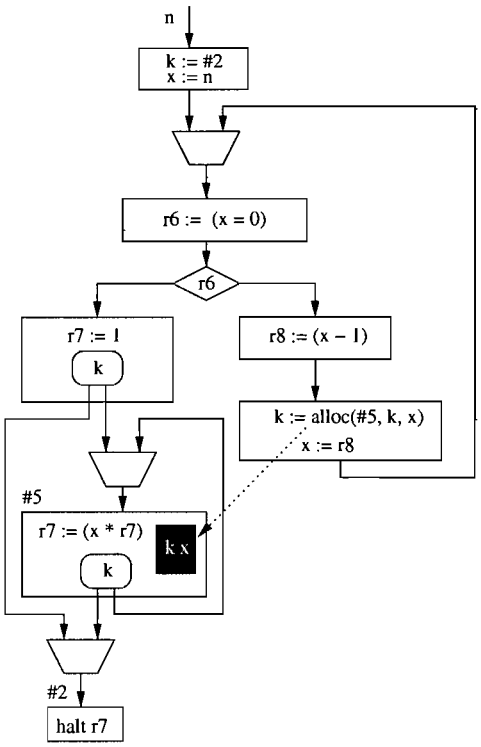


Fig. 7. CDFG for factorial

Figure 7 shows a Control and Data Flow Graph (CDFG) derived from the FP compiler's output. Processes flow through solid lines (data + control signals) while dotted lines denote closure allocations. Each rectangle represents an ASM device with a single state which performs basic calculations on its input (direct transfer, primitive operation or closure allocation) and sends the result to its output. Rectangles with rounded corners are function calls to closures: they send their input to either output depending on the closure's (k) tag bits.

This is illustrated in Figure 8, which represents the continuation to a recursive call to `fac` (closure #5); the continuation for this device may be the initial continuation (#2, output final result) or itself (#5), through merge nodes. At the top of the device is the closure memory. For an allocation (e.g. `alloc(#5, k, x)` in Figure 7), free variables are received on `alloc_data` and the corresponding address is sent back on `alloc_addr`. For a call, the address of the

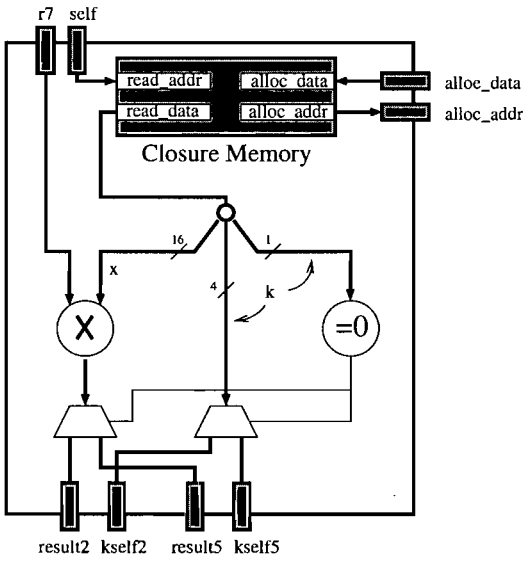


Fig. 8. Closure #5 device

free variables is received on `self`, with a result on `r7` since this is a continuation. With `self` directly connected to the closure memory, the corresponding variables (`x` and `k`) are automatically fetched. Then, `x` is multiplied with `r7` and the continuation (`k`) is called with the result. The call to `k` is represented by the multiplexers at the bottom of the figure: the tag part of `k` controls the multiplexers while it's address part flows through, along with the multiplier's result.

This is where CASM's high-level synchronization comes into play: the FP compiler's job is essentially limited to instantiating components and connecting them together; it is then guaranteed that for every (`self`, `r7`) pair received, a single output will be produced on either (`result2`, `kself2`) or (`result5`, `kself5`).

Also, the closure memory is an abstract component and different implementations can be created in minutes to fit a project's specific constraints (target technology, device size, required performance ...) An example of such an instantiation is shown in Figure 9. This one targets FPGA, with a priority on speed rather than space; power consumption *et al.* are non-issues. It implements a simple "free-list" algorithm with separate memories for closure data and pointers so that faster memories can be used (in LUTs, single-port, no read-on-write) and any transaction can be completed in a single cycle. Every pointer has one bit added to indicate a full memory: the last element of the free list is the only one to have this bit on. The pointer memory is instantiated on line 12 and it's initial value is taken from a file named "`initfile`". The `if` structure that starts on line 15 makes read (dealloc.) requests have a priority over allocation requests. Finally, lines 18 and 26 ensure that the "next" pointer is not changed until all other transactions have either completed or are ready to complete in the current cycle.

```
1. input  read_addr{"FS"}[AWIDTH]; //read
2. output read_data{"FS"}[CWIDTH];
3. input  alloc_data{"FS"}[CWIDTH]; //alloc
4. output alloc_addr{"FS"}[AWIDTH];
5.
6. define CWIDTH; //supplied on instantiation
7. define AWIDTH = pmem.width;
8. define MEM_DEPTH = 1 << AWIDTH;
9.
10. register nxfree[AWIDTH+1] = 0;
11. memory cmem{MEM_DEPTH}[CWIDTH];
12. memory pmem{MEM_DEPTH,"initfile"}[AWIDTH+1];
13.
14. asm {
15. S0: if(request(read_addr))
16.       read_data := cmem[read_addr];
17.       pmem[read_addr]:=nxfree.[AWIDTH-1..0];
18.       if(S0.complete)
19.         nxfree := 0::read_addr;
20.       end;
21.     elsif(request(alloc_data)
22.           && !nxfree.[AWIDTH])
23.       post addr = nxfree.[AWIDTH-1..0];
24.       cmem[addr] := alloc_data;
25.       alloc_addr := addr;
26.       if(S0.complete)
27.         nxfree := pmem[addr];
28.       end;
29.     end;
30.     goto S0;
31. }
```

Fig. 9. CASM source code for a closure memory

IV. Conclusion

Due to the recent technological evolution of computing hardware the problem of implementing a system has moved from mapping an algorithm to an existing architecture to that of constructing a special purpose architecture tailored to the algorithm. The goal of our work is to design hardware description languages that allow non-experts to easily implement high-performance provably correct systems by synthesizing them from a high level algorithmic description. The CASM language is inspired from the token machine paradigm where the system's operations are triggered by the presence of data. This gives a programming model where synchronization is implicit and it enables a high degree of parallelism between operations. CASM also supports recursion which is absent from all common HDLs.

Although CASM can be used for directly programming applications, we have also begun investigating its use as an intermediate representation for compiling functional programming (FP) languages to hardware. Indeed CASM excels at expressing data-driven dataflow systems that are conceptually related to FP languages. Our prototype FP compiler handles recursion and higher-order functions using the continuation-passing-style conversion and a combined control and data flow analysis (0-CFA). To maximize parallelism we use distributed memories and linked frames to implement the run time stacks of the processes. A functional closure is implemented by a device with a local memory containing the free-variables of the closure. Deallocation of closures is handled automatically in the case of continuation closures and must be performed explicitly by the programmer otherwise.

Once our compilers are fully operational we plan to evaluate their ease of use and performance by implementing realistic systems. Of particular interest is the speed and size of the resulting circuit compared to other HDLs. We anticipate some extensions to CASM and our FP language to improve their ease of use. Various optimizations are possible to the CASM compiler to minimize number of registers, delays, connections, etc. We also plan to target various back-ends including asynchronous designs, which seems to be the best known approach for designing low power circuits.

Our FP language will be extended with programmer declarations to allow the compiler to determine through analyses the width of data paths and closure memories (type annotations and type inference), the size of memories (maximal recursion depth and number of processes), and absence of deadlocks when multiple recursive processes are used. Our FP language also needs a way to link to devices written in other HDL (CASM, VHDL, etc.).

V. Acknowledgment

This work was supported by the National Sciences and Engineering Research Council of Canada (NSERC) under its Discovery Grant program.

References

[1] J. Dennis and D. Misunas, "A preliminary architecture for a basic data-flow processor," in *2nd ISCA*, January 1975, pp. 126–132.

[2] S. A. B. et all, *IEEE standard VHDL language reference manual, IEEE Std 1076-2002.* The Institute of Electrical and Electronics Engineers, Inc., May 2002.

[3] M. M. M. et all, *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001.* The Institute of Electrical and Electronics Engineers, Inc., September 2001.

[4] Celoxica, "HandleC Home page." [Online]. Available: http://www.celoxica.com/methodology/handelc.asp

[5] D. Ku and G. D. Micheli, "HardwareC: a language for hardware design," Computer Systems Laboratory, Stanford Univ., Stanford, Calif, Tech. Rep. SCSL/CSL/TR-90-419, August 1990.

[6] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," *Proc. Symp. on FPGAs for Custom Computing Machines*, pp. 136–144, April 1995.

[7] J. Zhu, D. D. Gajski, and R. Doemer, "Syntax and semantics of the spec C+ language, Tech. Rep. ICS-TR-97-16, 1997. [Online]. Available: citeseer.ist.psu.edu/article/zhu97syntax.html

[8] M. B. et all, "SystemC 2.0.1 Language Reference Manual," 2003.

[9] J. Lapalme and E. M. Aboulhamid, "eSys.Net Home page." [Online]. Available: http://www.esys-net.org

[10] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming.* ACM Press, 1998, pp. 174–184.

[11] J. J. O'Donnell, "From transistors to computer architecture: Teaching functional circuit specification in hydra," in *FPLE '95: Proceedings of the First International Symposium on Functional Programming Languages in Education.* Springer-Verlag, 1995, pp. 195–214.

[12] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Tech J. vol 34*, pp. 1045–1079, September 1955.

[13] E. F. Moore, "Automata Studies," *Annals of Mathematical Studies*, 1956.

[14] R. Kelsey, W. Clinger, and J. Rees, "Revised^5 Report on the Algorithmic Language Scheme." [Online]. Available: http://www.schemers.org/Documents/Standards/R5RS/

[15] J. Guy L. Steele, "Rabbit: A compiler for scheme," Tech. Rep., 1978.

[16] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," Ph.D. dissertation, 1991.

[17] A. W. Appel and T. Jim, "Continuation-passing, closure-passing style," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, 1989, pp. 293–302.

8.1 Modèle proprosé

Le compilateur Dynabit est présentement limité à l'implantation d'expressions simples en matériel sous forme de *pipeline*. Ce modèle engendre une sous-utilisation des ressources puisque le processeur n'est pas en mesure de fournir une donnée par cycle.

Afin d'améliorer la performance des communications, nous pourrions permettre au contrôleur de la zone dynamique d'accéder directement à la mémoire (DMA) sans intervention du processeur. Cette approche évite les communications inutiles avec le processeur, mais elle est tout de même limitée par le goulot d'étranglement engendré par les communications avec la mémoire.

Une autre approche possible consisterait à introduire la notion de récursivité, ce qui permettrait l'exécution d'une instruction dynamique sur plusieurs cycles, absorbant ainsi les coûts de communication. Avec la notion de récursivité, il est possible d'effectuer un compromis entre l'espace et la latence d'exécution d'une instruction spécialisée en variant le nombre de déroulements de boucles. Toutefois, des mécanismes de synchronisation doivent être introduits afin de coordonner le déplacement des données dans le système. De plus, contrairement à un *pipeline*, il est complexe de gérer plusieurs exécutions simultanées de l'instruction spécialisée (plus qu'une donnée présente dans la même instruction spécialisée).

La récusivité limitée aux appels en position terminale permet d'exprimer en matériel des structures de contrôle typiquement rencontrées dans le corps d'une fonction. Nos expériences nous ont permis d'observer qu'un circuit implantant une fonction de taille raisonnable occupe la majorité de la superficie de la zone dynamique. Ainsi nous croyons que la granularité des expressions compilées dynamiquement devrait être de la taille d'une fonction et que le compilateur doit supporter la récursivité limitée.

La récursion générale quant à elle permet l'implantation en matériel de toutes les structures de contrôle présentes dans les langages de programmation. Par contre, le concept de pile d'exécution (une ou plusieurs piles) doit exister ainsi que les mécanismes associés (dépassement de pile). Ce modèle s'applique aux architectures dont les zones dynamiques peuvent être plus larges.

8.2 Recherches futures

Nous croyons que les prochains axes de recherche en synthèse dynamique devraient être orientés sur l'intégration des langages de haut niveau. Nous avons démontré qu'il est possible d'effectuer les phases de la synthèse efficacement, et qu'ainsi la production de modules dynamiques est possible. Les concepts plus évolués peuvent s'implanter en matériel avec les primitives présentes dans Dynabit sans influencer les phases cruciales de la synthèse (e.g. placement et routage).

Les problèmes rencontrés par la synthèse de haut niveau s'apparentent aux problèmes auxquels font face les compilateurs classiques : il est souvent trop complexe de trouver la solution optimale, ce qui porte à recourir à des heuristiques pour déterminer une bonne implantation. Afin d'améliorer les choix effectués au sein de ses algorithmes, un compilateur dynamique se base sur l'information recueillie par profilage de l'application. La collecte de cette information engendre elle-même un coût en performance à considérer.

Idéalement, un compilateur dynamique est transparent et détermine automatiquement quelles sont les parties de code fréquemment exécutées qui pourraient tirer profit de l'accélération matérielle. Par conséquent, la gestion des instructions spécialisées ne devrait pas être visible à l'utilisateur.

Chapitre 9

Conclusion

Nos travaux de recherche ont un impact sur plusieurs aspects du domaine de l'exécution reconfigurable. Ils rendent possible la production et la manipulation dynamique de configurations sur un FPGA disponible commercialement. L'avancée technologique ainsi réalisée permet l'intégration d'un mode d'exécution innovateur qui combine l'évaluation partielle et la reconfiguration dynamique afin d'accélérer l'exécution de certaines applications autrefois limitées à une exécution sur un processeur d'usage général. Afin de démontrer le potentiel de ce mode d'exécution, nous avons réalisé une architecture matérielle reconfigurable ainsi qu'un compilateur dynamique capable de produire à la volée une configuration pour cette architecture.

Nous avons permis la production dynamique de configurations pour les FPGAs de Xilinx en déterminant la correspondance entre les bits de configuration et leur équivalent matériel. Afin de faciliter la manipulation de configurations, une bibliothèque a été créée afin d'abstraire ces correspondances. De nombreux outils ont été développés afin de pallier les lacunes des outils actuels principalement adaptés à la synthèse statique. Ce progrès ouvre la porte à la recherche sur le développement d'applications nécessitant une spécialisation dynamique. Nous croyons que le nombre d'applications dans cette catégorie croîtra dans les prochaines années vue la disponibilité d'une architecture facilitant leur déploiement et suite à la démonstration du gain en performance obtenu par une telle approche.

De plus, nos travaux de recherche ont démontré qu'il est possible d'effectuer de la synthèse à la volée sur les FPGAs actuels. Les temps de compilation sont de moins d'une seconde, ce

qui est plus rapide de plusieurs ordres de grandeur par rapport aux techniques usuelles. Cette accélération permet ainsi aux applications d'utiliser le concept d'évaluation partielle dans le cadre d'une exécution reconfigurable et permet une accélération substantielle de plusieurs applications en augmentant leur densité fonctionnelle.

Pour parvenir à des temps de synthèse aussi efficaces, nous avons développé des techniques de synthèse agressives qui acceptent de sacrifier certaines propriétés, telles que la densité, au profit de la vitesse d'exécution.

Quoique nos algorithmes de synthèse soient efficaces, nous sommes convaincus que de nouvelles améliorations pourront encore accélérer la synthèse. La recherche dans le domaine de la synthèse à la volée, qui n'en est qu'à ses balbutiements, mérite, selon nous, une plus grande attention de la part de l'industrie et du milieu académique. Des améliorations sur la conception des FPGAs, sur le développement d'architectures matérielles ou encore sur les techniques de compilation permettraient d'augmenter la classe d'applications supportées par l'exécution reconfigurable.

Le mode d'exécution que nous proposons s'apparente au modèle de co-design matériel où l'on enrichit le jeu d'instructions d'un processeur en lui ajoutant des unités de calcul spécialisées afin d'accélérer les opérations dans lesquelles est passée la majorité du temps d'exécution. Ainsi, les applications qui utilisent le co-design peuvent bénéficier de notre approche, puisqu'elle permet une spécialisation du jeu d'instructions adaptable à la nature des données au cours de l'exécution.

En conclusion, la synthèse à la volée est réalisable sur les FPGAs modernes et permet d'obtenir des gains en performance importants. Ce mode d'exécution se présente donc comme une sphère de recherche prometteuse, ce qui permettra d'obtenir un modèle d'exécution menant à l'accélération d'applications par l'utilisation du parallélisme massif disponible en matériel.

Bibliographie

[ABD92] Jeffrey M. Arnold, Duncan A. Buell, and Elaine G. Davis. Splash 2. In *SPAA '92 : Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, New York, NY, USA, 1992. ACM Press.

[AHS76] Ehud Artzy, James A. Hinds, and Harry J. Saal. A Fast Division Technique for Constant Divisors. *Communications of the ACM*, 19(2) :98–101, 1976.

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, Effective Dynamic Compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *Computer*, 26(3) :11–18, 1993.

[BCF+99] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : A Transparent Dynamic Optimization System. In *PLDI '00 : Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[BDFD08] Etienne Bergeron, Marc-André Daigneault, Marc Feeley, and Jean Pierre David. Using Dynamic Reconfiguration to Implement High-Resolution Programmable Delays on an FPGA. *Workshop on Circuits and Systems and TAISA Conference, NEWCAS-TAISA*, pages 265–268, June 2008.

[BFD07] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Toward On-Chip JIT Synthesis on Xilinx Virtex-II Pro FPGAs. In *50th Internationall Midwest Symposium on Circuits and Systems/5th International Northeast Workshop on Circuits (MWCAS/NEWCAS), Montréal, Canada*, August 2007.

[BFD08] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs. In *Compiler Conference*, pages 178–192, 2008.

[BH94] Preston Briggs and Tim Harvey. Multiplication by Integer Constants. Technical report, Rice University, July 1994.

[BSMFD05] Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley, and Jean Pierre David. High-Level Synthesis for Data-Driven Applications. In *IEEE International Workshop on Rapid System Prototyping*, pages 54–60, 2005.

[CAK+81] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for Repetitive Transactions and Ad Hoc Queries In System R. *ACM Transactions on Database Systems*, 6(1) :70–94, 1981.

[Cal02] Timothy John Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures.* PhD thesis, University of California, Berkeley, 2002.

[CCDW98] Timothy J. Callahan, Philip Chong, Andre Dehon, and John Wawrzynek. Fast Module Mapping and Placement for Datapaths in FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, 1998.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and Andre DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *FPL '00 : Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications* , pages 605–614, 2000.

[CHW00] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4) :62–69, 2000.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and*

Applications (OOPSLA), volume 24, pages 49–70, New York, NY, 1989. ACM Press.

[CWG+98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55–64, Monterey, CA, 1998.

[DB04a] Jean Pierre David and Etienne Bergeron. A Step Towards Intelligent Translation from High-Level Design to RTL. In *IWSOC : International Workshop System-on-Chip*, pages 183–188, 2004.

[DB04b] Jean Pierre David and Etienne Bergeron. An Intermediate Level HDL for System Level Design. *Forum on Specification and Design Languages*, September 2004.

[DeH94] André DeHon. DPGA-Coupled Microprocessors : Commodity ICs for the Early 21st Century. In Duncan A. Buell and Kenneth L. Pocek, editor, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[Deh96] Andre Maurice Dehon. *Reconfigurable architectures for general-purpose computing.* PhD thesis, Massachusetts Institute of Technology, 1996.

[DJSM98] Michel Dubois, Jaeheon Jeong, Yong Ho Song, and Adrian Moga. Rapid hardware prototyping on rpm-2. *Design and Test of Computers, IEEE*, 15(3) :112–118, Jul-Sep 1998.

[EBTB63] Gerald Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, EC-12(5) :747–755, December 1963.

[EH94a] James G. Eldredge and Brad L. Hutchings. Density enhancement of a neural network using fpgas and run-time reconfiguration. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, 1994.

[EH94b] James G. Eldredge and Brad L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[EMHB95] C. Ebeling, L. McMurchie, S.A. Hauck, and S. Burns. Placement and Routing Tools for the Triptych FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4) :473–482, Dec 1995.

[Est02] Gerald Estrin. Reconfigurable Computer Origins : the UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing*, 24(4) :3–9, 2002.

[FAM+97] J. Faura, M. Aguirre, J. Moreno, P. van Duong, and J. Insenser. FIPSOC : A Field Programmable System On a Chip. In *Design of Circuits and Integrated Systems*, 1997.

[Fee] Marc Feeley. Gambit Home Page. `http://www.iro.umontreal.ca/~gambit/`.

[GHK+90] Maya Gokhale, William Holmes, Andrew Kopser, Dick Kunze, Daniel P. Lopresti, Sara Lucas, Ronald Minnich, and Peter Olsen. SPLASH : A Reconfigurable Linear Logic Array. In *Proceedings of the International Conference on Parallel Processing*, pages 526–532, 1990.

[GMP+97] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 163–178. ACM, June 1997.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In *FPGA '04 : Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 162–170, New York, NY, USA, 2004. ACM.

[GS05] Brian Greskamp and Ron Sass. A Virtual Machine for Merit-Based Runtime Reconfiguration. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 287–288, April 2005.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench : A Reconfigurable Architecture and Compiler. *Computer*, 33(4) :70–77, 2000.

[HLK02] Edson L. Horta, John W. Lockwood, and Sérgio T. Kofuji. Using parbit to implement partial run-time reconfigurable systems. In *FPL '02 : Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on*

Field-Programmable Logic and Applications, pages 182–191, London, UK, 2002. Springer-Verlag.

[HW97] John R. Hauser and John Wawrzynek. GARP : A MIPS Processor with a Reconfigurable Coprocessor. In J. Arnold and K. L. Pocek, editor, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, April 1997.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[JTS05] Kimmo Järvinen, Matti Tommiska, and Jorma Skyttä. Hardware implementation analysis of the md5 hash algorithm. *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05), 2005*, 9 :298a–298a, Jan. 2005.

[KdlTRJ06] Y.E. Krasteva, E. de la Torre, T. Riesgo, and D. Joly. Virtex II FPGA Bitstream Manipulation : Application to Reconfiguration Control Systems. *International Conference on Field Programmable Logic and Applications, 2006. FPL '06*, pages 1–4, August 2006.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A Case for Runtime Code Generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[KEH93] D. Keppel, S. J. Eggers, and R. R. Henry. Evaluating Runtime-Compiled Value-specific Optimization. Technical Report TR-93-11-02, University of Washington Department of Computer Science and Engineering, 1993.

[KJdlTR05] Yana E. Krasteva, Ana B. Jimeno, Eduardo de la Torre, and Teresa Riesgo. Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs. In *RSP '05 : Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 77–83, Washington, DC, USA, 2005. IEEE Computer Society.

[LD97] Mark Leone and R. Kent Dybvig. Dynamo : A Staged Compiler Architecture for Dynamic Program Optimization. Technical report, Department of Computer Science, Indiana University, September 1997. Technical Report #490.

[LGvI+97] David M. Lewis, David R. Galloway, Marcus van Ierssel, Jonathan Rose, and Paul Chow. The Transmogrifier-2 : A 1 Million Gate Rapid Prototyping System. In *FPGA '97 : Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*, pages 53–61, New York, NY, USA, 1997. ACM Press.

[LL93] Mark Leone and Peter Lee. Deferred Compilation : The Automation of Run-Time Code Generation. Technical report, Carnegie Mellon University, 1993.

[LL94] Mark Leone and Peter Lee. Lightweight Run-Time Code Generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[LL96] Mark Leone and Peter Lee. A Declarative Approach to Run-Time Code Generation. In *Workshop on Compiler Support for System Software (WCSSS)*, pages 8–17, February 1996.

[LMS06] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set.* CRC Press, Inc., Boca Raton, FL, USA, 2006.

[LSV06] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM Transactions on Design Automation of Electronic Systems*, pages 659–681, July 2006.

[LV04] Roman Lysecky and Frank Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In *DATE '04 : Proceedings of the conference on Design, automation and test in Europe*, page 10480, Washington, DC, USA, 2004. IEEE Computer Society.

[LVT04] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA Routing for Just-in-Time FPGA Compilation. In *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 954–959. IEEE Computer Society, 2004.

[LVT05] R. Lysecky, F. Vahid, and S. X.-D. Tan. A Study of the Scalability of On-Chip Routing for Just-In-Time FPGA Compilation. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 57–62, 2005.

[Mat00] Matthew C. Merten and Andrew R. Trick and Erik M. Nystrom and Ronald D. Barnes and Wen-mei W. Hmu. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *ISCA '00 : Proceedings of the 27th annual*

international symposium on Computer architecture, pages 59–70, New York, NY, USA, 2000. ACM.

[ME95] Larry McMurchie and Carl Ebeling. PathFinder : a negotiation-based performance-driven router for FPGAs. In *FPGA '95 : Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117, New York, NY, USA, 1995. ACM.

[Mir87] Eliot Miranda. BrouHaHa – A Portable Smalltalk Interpreter. In *OOPSLA '87 : Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 354–365, New York, NY, USA, 1987. ACM Press.

[Pic] Jim Pick. The Kaffe homepage. `http://www.kaffe.org/`.

[PJA+99] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, Andre DeHon, and John Wawrzynek. Embedded DRAM for a Reconfigurable Array. In *IEEE Symposium on VLSI Circuits*, pages 145–148, Kyoto, Japan, 1999.

[PLR85] Rob Pike, Bart Locanthi, and John Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software – Practice and Experience*, 15(2) :131–151, 1985.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[SBR98] Jordan S. Swartz, Vaughn Betz, and Jonathan Rose. A Fast Routability-Driven Router for FPGAs. In *FPGA '98 : Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 140–149, New York, NY, USA, 1998. ACM.

[SMFD06] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard : A Scheme to Hardware Compiler. *Scheme and Functional Programming Workshop*, pages 39–49, 2006.

[Sun99] Sun. The Java Hotspot Performance Engine Architecture. Technical report, Sun Microsystems, April 1999.

[Sun02] Sun. The Java HotSpotTM Virtual Machine, v1.4.1. Technical report, Sun Microsystems, September 2002.

[SWM+02] H. Schmit, D. Whelihan, M. Moe, B. Levine, and R. Taylor. PipeRench : A Virtualized Programmable Datapath, 2002.

[TCE+95] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. A First Generation DPGA Implementation. In *FPD'94 - Third Canadian Workshop of Field-Programmable Devices*, pages 138–143, May 1995.

[TGP07] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident : From High-Level Language to Hardware Circuitry. *Computer*, 40(3) :28–37, 2007.

[Tho68] Ken Thompson. Regular Expression Search Algorithm. *Communcations of the ACM*, 11(6) :419–422, 1968.

[VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories : Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1) :56–69, 1996.

[VMS97] John Villasenor and William H. Mangione-Smith. Configurable Computing. *Scientific American*, pages 66–71, June 1997.

[WH97] M. J. Wirthlin and B. L. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, Monterey, CA, 1997.

[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software : Raw Machines. *Computer*, 30(9) :86–93, 1997.

[Xil02] Xilinx. Two Flows for Partial Reconfigurable Core Based on Small Bit Manipulations. Technical report, Xilinx, September 2002.

Annexe A

Benchmarks

```
(include "crypto.scm")

(define make-md5-partial-bruteforce
  (lambda (prefix hash)

    (define word  (make-u32vector 16))
    (define H     (make-u32vector 4))
    (define plen  (string-length prefix))

    (define F (lambda (i x y z)
                (cond
                 ((< i 16)  (#%or (#%and x y) (#%and (#%not x) z)))
                 ((< i 32)  (#%or (#%and z x) (#%and (#%not z) y)))
                 ((< i 48)  (#%xor x y z))
                 (else      (#%xor y (#%or x (#%not z)))))))

    (define K (lambda (i)
                (inexact->exact (floor (* (abs (sin (+ i 1))) (expt 2 32))))))

    (define R (lambda (i)
                (cond ((< i 16) (list-ref '(7 12 17 22) (modulo i 4)))
                      ((< i 32) (list-ref '(5  9 14 20) (modulo i 4)))
                      ((< i 48) (list-ref '(4 11 16 23) (modulo i 4)))
                      (else     (list-ref '(6 10 15 21) (modulo i 4)))))))

    (define W (lambda (i nc)
                (let ((offset (modulo (cond ((< i 16) i)
                                            ((< i 32) (+ (* 5 i) 1))
                                            ((< i 48) (+ (* 3 i) 5))
                                            (else (* 7 i)))
                                      16)))
                  (if (= offset (quotient plen 4))
                      nc
                      (u32vector-ref word offset)))))

    (let ((num (string->number hash 16)))      ;; Initialisation
      (u32vector-set! H 3 (u32->little-indian num))
      (u32vector-set! H 2 (u32->little-indian (arithmetic-shift num -32)))
      (u32vector-set! H 1 (u32->little-indian (arithmetic-shift num -64)))
      (u32vector-set! H 0 (u32->little-indian (arithmetic-shift num -96))))

    (u32vector-set! word 14 (* (+ 4 plen) 8))

    (let loop ((i 0)                           ;; Padding
               (lst (append (map char->integer (string->list prefix))
                            '(0 0 0 0 #x80))))
      (let ((offset (quotient i 4))
            (suboff (modulo i 4)))
        (if (null? lst)
            word
            (begin
              (u32vector-set! word offset
                              (+ (u32vector-ref word offset)
                                 (* (car lst)
                                    (list-ref '(#x1 #x100 #x10000 #x1000000)
                                              suboff))))
              (loop (+ i 1) (cdr lst))))))))
```

```
(lambda (nc)                                        ;; Partial MD5
  (let ((h0 #x67452301)
        (h1 #xEFCDAB89)
        (h2 #x98BADCFE)
        (h3 #x10325476))
    (define iter
      (lambda (i a b c d nc)
        (if (= i 64)
            (and (= a (#%sub (u32vector-ref H 0) h0))
                 (= b (#%sub (u32vector-ref H 1) h1))
                 (= c (#%sub (u32vector-ref H 2) h2))
                 (= d (#%sub (u32vector-ref H 3) h3)))
            (let ((na (#%add b (#%rol (#%add a (F i b c d)
                                              (K i) (W i nc))
                                      (R i)))))
              (iter (+ i 1) d na b c nc)))))
    (iter 0 h0 h1 h2 h3 nc))))
```

```
(define add1
  (lambda (x)
    (+ x 1)))

(define mean4
  (lambda (a b c d)
    (quotient (+ a b c d) 4)))

(define max4
  (lambda (a b c d)
    (let* ((m a)
           (m (if (< m b) b m))
           (m (if (< m c) c m))
           (m (if (< m d) d m)))
      m)))

(define upcase
  (lambda (c)
    (if (and (<= 97 c) (<= c 122))
        (- c 32)
        c)))

(define isqrt
  (lambda (x)

    (define loop
      (lambda (op res one count)
        (if (= count 0)
            res
            (let* ((s    (+ res one))
                   (c    (<= s op))
                   (nop  (if c (- op s) op))
                   (nres (if c (+ s one) res)))
              (loop nop (#%shr nres 1) (#%shr one 2) (- count 2))))))

    (loop x 0 (#%shl 1 30) 32)))

(define bit-count
  (lambda (w)
    (let* ((a (#%and w #x55555555))
           (b (#%shr (#%and w #xAAAAAAAA) 1))
           (s (#%add a b)))
      (let* ((a (#%and s #x33333333))
             (b (#%shr (#%and s #xCCCCCCCC) 2))
             (s (#%add a b)))
        (let* ((a (#%and s #x0F0F0F0F))
               (b (#%shr (#%and s #xF0F0F0F0) 4))
               (s (#%add a b)))
          (let* ((a (#%and s #x00FF00FF))
                 (b (#%shr (#%and s #xFF00FF00) 8))
                 (s (#%add a b)))
            s))))))
```

```
(define bit-rev
  (lambda (w)
    (let* ((a (#%and w #xFFFF0000))
           (b (#%and w #x0000FFFF))
           (s (#%or (#%shl b 16)
                    (#%shr a 16))))
      (let* ((a (#%and s #xFF00FF00))
             (b (#%and s #x00FF00FF))
             (s (#%or (#%shl b 8)
                      (#%shr a 8))))
        (let* ((a (#%and s #xF0F0F0F0))
               (b (#%and s #x0F0F0F0F))
               (s (#%or (#%shl b 4)
                        (#%shr a 4))))
          (let* ((a (#%and s #xCCCCCCCC))
                 (b (#%and s #x33333333))
                 (s (#%or (#%shl b 2)
                          (#%shr a 2))))
            (let* ((a (#%and s #xAAAAAAAA))
                   (b (#%and s #x55555555))
                   (s (#%or (#%shl b 1)
                            (#%shr a 1))))
              s)))))))

(define bit-lg2
  (lambda (w)
    (let* ((w (#%or w (#%shr w 1)))
           (w (#%or w (#%shr w 2)))
           (w (#%or w (#%shr w 4)))
           (w (#%or w (#%shr w 8)))
           (w (#%or w (#%shr w 16))))
      (bit-count* w))))

(define fib-n
  (lambda (n)
    (define f
      (lambda (n a b)
        (if (= n 0)
            b
            (f (- n 1) b (+ a b)))))
    (lambda (a b)
      (f n a b))))
```

```
(define hash-iter
  (lambda (depth va vb vc vd)
    (if (=  depth 0)
        (+ va vb vc vd)
        (hash-iter (- depth 1)
                   vd
                   (#%xor (#%add va vb) vc)
                   vb
                   (#%xor (#%add vc vd) va)))))

(define hash
  (lambda (depth)
    (lambda (a b c d)
      (hash-iter depth a b c d))))


(define hashr-iter
  (lambda (depth va vb vc vd)
    (if (=  depth 0)
        (+ va vb vc vd)
        (hashr-iter (- depth 1)
                    vd
                    (#%rol (#%xor (#%add va vb) vc) depth)
                    vb
                    (#%xor (#%add vc vd) va)))))

(define hashr
  (lambda (depth)
    (lambda (a b c d)
      (hashr-iter depth a b c d))))


(define make-prng
  (lambda (a c m)
    (lambda (x)
      (modulo (+ (* a x) c) m))))
```